Storage Requirement Estimation for Data Intensive Applications with Partially Fixed Execution Ordering

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 partly fixed. At that stage, none of the existing estimation tools are adequate, as they either assume a fully specified execution order or ignore it completely. Using a representative application demonstrator, we show how our technique can effectively guide the designer to achieve a transformed specification with low storage requirement.

1. INTRODUCTION

Many embedded HW/SW 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 [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. 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 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 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 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 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 this paper we propose a new technique for estimating the storage requirements for data intensive applications with a partially fixed execution ordering. The methodology is partly based on previous work done by Florin Balasa et al. [1], and uses a polyhedral model of the arrays and the execution ordering. Several major extensions are proposed here to achieve our goals. The rest of this paper starts with a presentation of previous work on storage requirement estimation, including the contribution of Balasa et al. This is followed by the description of the new technique in section 3 and by experimental results on a representative application demonstrator in section 4. 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, 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 [6]. Techniques such as clique partitioning are also exploited to group variables that can be mapped together [7]. A good introduction to the scalar-based storage unit estimation can be found in [4]. 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 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. In [8] a production time axis is created for each array. This models the relative production and consumption time, or date, of the individual array accesses. The maximum difference between the production and consumption date found for any two depending instances, gives the storage requirement for this array. The total storage requirement is the sum of the requirements for each array. To generate the production time axis, the execution ordering has to be fully fixed. Also, since each array is treated separately, only in-place mapping internally to an array is considered, not the possibility of mapping arrays in-place of each other. Another approach is taken in [5]. 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. The total storage requirement equals the peak bounding rectangle. If the difference between the upper and lower bounds for this critical rectangle is too large, 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. [9] describes a methodology for so-called exact memory size estimation for array computation. It is based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. 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.

In contrast to the methods described in the previous paragraph, the storage requirement estimation technique presented by Balasa et al. in [1] does not require the execution ordering to be fixed. On the contrary, it does not take execution ordering into account at all. They start with an extended data-dependency analysis where not only dependencies between array accesses in the code are taken into account, but also *which parts* of an array produced by one instruction that are read by another (or possibly the same) instruction. For each instruction in the code, a definition domain is extracted, containing each array element produced by the instruction. Similarly, operand domains are extracted for the parts of arrays read by the instruction. Through an analytical partitioning of the arrays involving, among other steps, intersection of these domains, they end up with a number of non-overlapping *basic sets* and the dependencies between them. Array elements common to a given set of domains and only to them constitute a basic set. The domains and basic sets 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. This information is used to generate a data-flow graph where the basic sets are the nodes and the dependencies between them are the branches. The total storage requirement for the application is found through a traversal of this graph, where basic sets are selected for production by a greedy algorithm. A basic set is ready for production when all basic sets it depends on have been produced and is consumed when the last basic set depending on it has been produced. Further details of the traversal are outside the scope of this paper. The maximal combined size of simultaneously alive basic sets gives the storage requirement.

```
L.1         (j: 0 .. 3)::
              (k: 0 .. 2)::
              begin
I.1             B[0][j][k] = f( A[j][k] );
              end;

L.2         (i: 1 .. 5)::
              (j: 0 .. 3)::
                (k: 0 .. 2)::
                begin
I.2               B[i][j][k] = g( B[i-1][j][k] );
I.3               C[i][j][k] = if (j >= 2) -> h( B[i][j-2][k] ) fi;
I.4               D[j][k] = if (i >= 5) ->  l( B[5][j][k] ) fi;
                end;
```

Figure 1: Code example (Silage language)

Since the rest of this paper in part is based on this work, we now present a simple illustrative example in the single-assignment language Silage. Note that very small loop bounds are used here to allow a graphical representation, but the mathematics used in the techniques and tools result in a complexity that is nearly independent of the actual loop bounds. Let us focus on the accesses to array B[u][v][w] in Figure 1. The array elements are produced by instructions I.1 and I.2, and read (and finally consumed) by instructions I.2, I.3, and I.4. Figure 2 gives examples of the LBL descriptions of the production and reading of array B[u][v][w] as found using the methodology from [1].

Instruction I.2: Definition domain (elements produced)	$B\begin{bmatrix}u\\v\\w\end{bmatrix}=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \Bigg\vert \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&1\\0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}\geq\begin{bmatrix}1\\-5\\0\\-3\\0\\-2\end{bmatrix}$
Instruction I.3: Operand domain (elements read)	$B\begin{bmatrix}u\\v\\w\end{bmatrix}=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}+\begin{bmatrix}0\\-2\\0\end{bmatrix} \Bigg\vert \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&1\\0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}\geq\begin{bmatrix}1\\-5\\2\\-3\\0\\-2\end{bmatrix}$

Figure 2: LBL descriptions for definition and operand domains

The part of array B[u][v][w] that is produced by instruction I.1 is read and only read, and thus also consumed, by instruction I.2. It therefore results in one basic set only, B(0). The part of array B[u][v][w] produced by instruction I.2, however, is read by instruction I.2, I.3, and/or I.4. Through intersection of the different definition and operand domains this part of array B[u][v][w] is split into several basic sets depending on how they interrelate with each other. Some of the elements are for instance read by I.2 and I.3 and not I.4, constituting basic set B(2). Figure 3 gives a graphical description of the basic sets for the B[u][v][w] and C[u][v][w] arrays. Figure 4 gives an example of the LBL description for one of the basic sets, B(2).

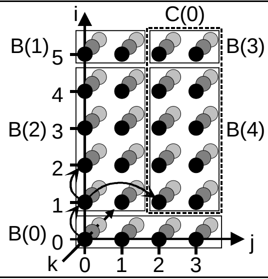

Figure 3: Iteration space for array B[u][v][w] and C[u][v][w] in the code example in Figure 1. Dependency shown for B[1][0][0].

$$B(2): B\begin{bmatrix}u\\v\\w\end{bmatrix}=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix} \Bigg\vert \begin{bmatrix}1&0&0\\-1&0&0\\0&1&0\\0&-1&0\\0&0&1\\0&0&-1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}\geq\begin{bmatrix}1\\-4\\0\\-1\\0\\-2\end{bmatrix}$$

Figure 4: LBL description for basic set B(2) of array B[u][v][w]

When the basic sets and the dependencies between them for all arrays in the code are found, the data-flow graph of Figure 5 is generated. The annotations to the nodes are the names and sizes of the corresponding basic sets, while the annotations to the branches are the dependency sizes. During the traversal of this graph, it is detected that the maximal combined size of simultaneously alive basic sets is reached when basic sets C(0), B(1) and B(3) are alive. Together they require 30+6+6=42 storage locations. Note that B(4) only requires 6 locations since it is mainly self-dependent.

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 described permit the designer to specify partial ordering constraints, which is really essential during the early exploration of the code transformations.

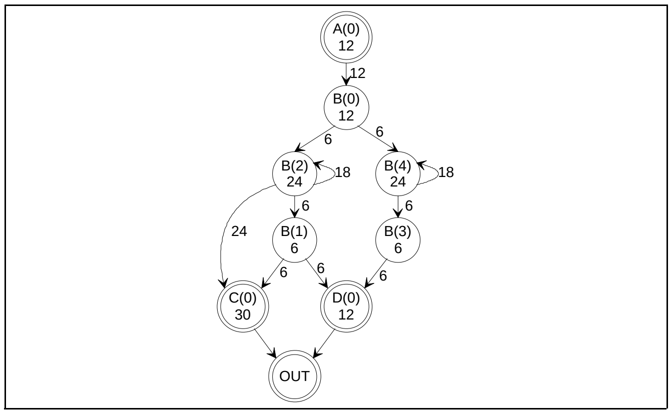

Figure 5: Data-flow graph for the code example in Figure 1.

3. ESTIMATION WITH PARTIALLY FIXED EXECUTION ORDERING

Look at the basic sets B(2) and C(0) in Figure 5 and the dependency between them. The size of basic set B(2) is 24. This corresponds to its worst-case storage requirement needed if the whole of B(2) is produced before the first element of C(0). As can be seen from Figure 3, this will happen if for example k is the innermost loop, followed by i, and with j as the outermost loop. As Figure 3 also shows, the best-case storage requirement is 2, achieved if j and k are interchanged in the above ordering. Then only elements B[1][0][0] and B[1][1][0] are produced before element C[1][2][0], which can potentially be mapped in-place with B[1][0][0]. Assume now that early in the design trajectory, we want to explore the consequences of having k as the outermost loop. Through inspection of Figure 3 we find that the dependency between B(2) and C(0) does not cross the k-dimension. Accordingly all elements produced for one value of k are also consumed for this value of k. Consequently the worst-case and best-case storage requirements are now 8 and 2 respectively, indicating that it is indeed advantageous to place k outermost. It may not be this simple however, since there can be conflicting dependencies in other parts of the code, giving rise to the need of an automated estimation tool to be able to take everything into account. This also rationalizes why we need to be able to do estimations on a partially fixed execution ordering. With estimation tools requiring the execution ordering to be fully specified, every alternative ordering with k as the outermost loop would have to be explored. It also shows the shortcomings of not considering the execution ordering at all, as in [1], since it results in a large overestimate.

Our methodology is useful for a large class of applications. There are certain restrictions 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 (achievable by a good array data-flow analysis preprocessing). Also array elements in a basic set must be produced and read sequentially, implying that a vector between depending elements in two basic sets have the same direction and length for all depending elements in the basic sets. The last requirement entail that if array element B[1][2][0] depends on A[0][0][0], then B[1][3][0] must depend on A[0][1][0] if the A-elements and B-elements belong to the same two basic sets. If this is not the case, the non-uniform basic sets can be split.

3.1 Basic Set and Dependency Sizes

The starting point for our estimation methodology is the basic set and dependency information as described in the last part of Section 2. The main principles can be used on any polyhedral description of sets of signals and their dependencies though. To overcome the problem of overestimation found in [1], we exploit information about the partly fixed execution ordering. Upper and lower bounds on the number of simultaneously alive elements in the dependencies between basic sets are calculated. This corresponds to the number of elements produced in one basic set before the first of them (potentially) can be overwritten by an element produced in the depending basic set. The sizes of the dependencies are in turn used to calculate new basic set sizes. The algorithm for estimating the upper and lower bounds of the dependency sizes is given in Figure 6. A branch in the data-flow graph contains information concerning what part of a basic-set the other basic set depends on, called the Dependency Part (DP). The first step in the algorithm is the orthogonalization of the DP. Each dimension is extended as needed until all dimensions are orthogonal with respect to each other, see Figure 7 for a two-dimensional example. The orthogonalization is necessary due to the possibly complex shapes of the DPs, which do not lend themselves easily to the type of calculations included in our estimation techniques. As can be seen it instigates a potential overestimate, but these acceptable errors are the price we have to pay for keeping the estimation complexity at a reasonable level. We are currently working on techniques to allow triangular forms in the DPs. When we in the sequel use the word lower bound, we mean the bound after this approximation.

```
void EstimateDependencySizes( data-flow graph )
for all basic sets B in data-flow graph {
  for all dependencies to basic sets $B_D$ depending on B {
    calculate the orthogonal DP, DV, and DVP of B with respect to $B_D$
    remove from the DP elements where for all SD the values are
                                                  larger than their SV
    define set UnspecifiedSpanningDimensions (USD) = (all SD)
    define set SpecifiedSpanningDimensions (SSD) = Ø
    for each specified dimension $d_i$ {
      if specification starts from innermost dimension {
        expand the dimension $d_i$ of the DVP to the border of DP
        if ($d_i \notin$ USD)
          remove from DVP elements that may not be visited
        else {
          remove from DP elements that will not to be visited
          USD = USD - $d_i$
          SSD = SSD + $d_i$
          if(USD = Ø)
            DP = DVP
      } }
      else {
        if ($d_i \notin$ USD)
          if (SSD = Ø)
            remove $d_i$ from DP
          else
            remove from DP elements that will not to be visited
        else {
          remove from DP elements that will not to be visited
          expand DVP with elements that will to be visited
          USD = USD - $d_i$
          SSD = SSD + $d_i$
          if(USD = Ø)
            DP = DVP
    } } }
    update data-flow graph(dependency upper bound = size(DP),
                               dependency lower bound = size(DVP))
} }
```

Figure 6: Pseudo-code for dependency size estimation algorithm

The code requirement of sequential production and reading allows us to define one element in the DP as the "smallest" element that will always be produced first regardless of the chosen execution ordering. For basic set B(2) in Figure 3 this is element B[1][0][0]. The estimation algorithm generates a Dependency Vector (DV) from this smallest element of the DP to the depending element in the depending basic set. For the dependency between B(2) and C(0) in Figure 3, the DV will thus start at (i,j,k)-point (1,0,0) and end at (1,2,0). The DV cover all or (more typically) a subset of the dimensions in the iteration space, defined as Spanning Dimensions (SD). We also define the values of the end points of the DV as the Spanning Values (SV). Since the DV in our example only covers the j-dimension, we have only one SD and one SV, the j-dimension and 2 respectively. The SD spans a polytope (partly) overlapping the DP. After intersection with the DP this polytope is denoted the Dependency Vector Polytope (DVP). Again referring to Figure 3, the DVP for the dependency between B(2) and C(0) consists of two elements, as shown in Figure 8. Note that for comparison with the graphical description a three-dimensional LBL is used, while the DVP really only has one dimension, the j-dimension.

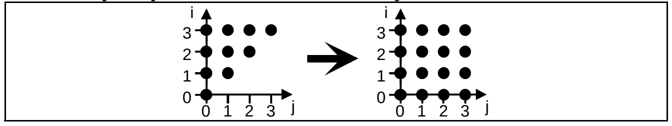

Figure 7: Orthogonalization of Dependency Part

We are now ready to start estimating the upper and lower bounds on the dependency sizes. If no execution order information is available, the lower bound can be found directly from the size of the DVP. No matter what execution order is chosen in the end, the elements of this polytope will always be produced before any elements of the depending basic set are produced. If there is no overlap between the depending basic set and the DP, also the upper bound can be found directly since it then equals the size of the DP. If overlap exists, we can first remove from the DP the elements that, for all SD, have larger values then the corresponding SV, since they can not possibly be produced before the first element in the depending basic set.

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

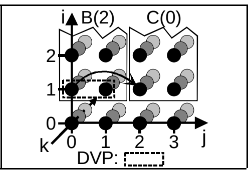

Figure 8: Dependency Vector Polytope

As the execution order is gradually fixed, the size of the DVP will be extended, and/or the size of the DP will be reduced. In most cases they will coalesce when all the spanning dimensions are fixed. When this does not happen, it is caused by simplifications in the extension/removal of elements in the DP and DVP to lower the estimation complexity. Assume now that the k-dimension is specified as the innermost dimension. According to the algorithm in Figure 6 the DVP is extended to encompass the full length of the k-dimension as shown in Figure 9. The DP and with it the upper bound on the dependency size, do not change. If now the j-dimension is specified as the second innermost dimension, this dimension of the DVP is to be expanded till the border of the DP. For our example, this is already the case, so the DVP does not change. Since the j-dimension is the last spanning dimension, the DVP size is also the upper bound on the dependency size.

The execution ordering may as well be specified starting with the outermost dimension. With i outermost, we can remove it from the DP since it is not a spanning dimension, and we get a new DP as shown in Figure 10. Knowing that i is non-spanning dimension also ascertain that the DVP does not change, so we get a reduced upper bound and an unchanged lower bound on the dependency size. With j as second outermost dimension we have to expand the remaining unspecified dimensions of the DVP till the DP border. For our example this only applies to the k-dimension, and we are back to the DVP and lower bound of Figure 9. Table 1 summarizes the estimation results for the different partly fixed execution orderings.

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

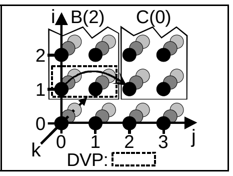

Figure 9: Dependency Vector Polytope with k innermost

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

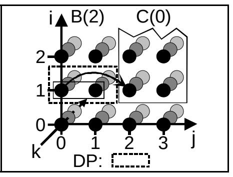

Figure 10: Dependency Part with i as outermost dimension

Fixed Dimension(s)		Upper bound	Lower bound
Outermost	Innermost		
None		24	2
	k	24	6
	j,k	6	6
i		6	2
i,j		6	6

Table 1: Dependency size estimat results

4. ESTIMATION ON 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 [2] for a more detailed description of this part of the standard. We will now use this real life application to show how storage requirement estimation can be used during the design trajectory. A part of the code is given in Figure 11 with the corresponding data-flow graph in Figure 12.

The loops in the upper nest of Figure 11 can be interchanged in 4!=24 ways. Because of the symmetry in the loops only 6 of these alternatives have to be investigated, see Table 2; interchange between y_s & x_s OR y_p & x_p leads to the same data in-place mapping opportunity.

```
(y_s : 0 .. 31 )::
 (x_s : 0 .. 31 )::
  (y_p : 0 .. 15 )::
   (x_p : 0 .. 15 )::
       sad[y_s][x_s][y_p][x_p] =
S.1     if ((x_p == 0)&(y_p == 0)) ->
                 f(curr[y_p][x_p], prev[y_s+y_p][x_s+x_p])
S.2     || ((x_p == 0)&(y_p != 0)) ->
                g(sad[y_s][x_s][y_p-1][15], curr[y_p][x_p],
                                     prev[y_s+y_p][x_s+x_p])
S.3     || 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])
        fi;

(y_s : 0 .. 31 )::
    (x_s : 0 .. 31 )::
S.4     result[y_s][x_s] = h(sad[y_s][x_s][15][15]);
```

Figure 11: MPEG-4 motion estimation kernel (Silage language)

Assuming that the prev[][] array is already in memory from the previous calculation, we have to investigate how the execution ordering influences the size of the sad[][][][] and curr[][] arrays. Given an external restriction that the curr[y_p][x_p] pixels are presented sequentially and row first (curr[0][0], curr[0][1], ...

curr[0][15], curr[1][0] ...) at the input, the required storage for the curr[][] array can be made as small as possible if we use alternative 2) from Table 2. Indeed, all calculations for each pixel can be completed before the next pixel arrives, and we need only one storage location for the curr[][] array. In any of the other methods, we have to buffer the curr[][] elements, since they are needed in bursts. This will require 16x16=256 storage locations. As a first design step it is therefore natural to investigate what the storage requirement for the sad[][][][] array is, if the execution order is optimized for the curr[][] array, beginning with the ordering of y_p as the outermost loop. The estimation results are listed in Table 3. The last row contains the number of simultaneously alive array elements found through the traversal of the data-flow graph. Because of the loop between sad(3), sad(4), and sad(5) in Figure 12, parts of these basic sets may in the worst case be alive simultaneously. Even in the best case, when this is not the situation, the increase in the lower bound on the storage requirement (1025-1 = 1024) exceeds the storage requirement for all of the curr[][] array (256). Consequently we can rule out any interchange alternative with y_p as the outermost loop. We are then left with interchange method 1), 3), and 5), each with y_s as the outermost loop.

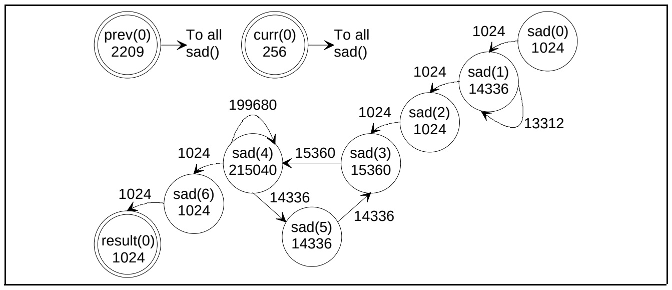

Figure 12: Data-flow graph for MPEG-4 motion estimation kernel

	↓ Outermost			Innermost ↓	
1)	y_s	x_s	y_p	x_p	←original
2)	y_p	x_p	y_s	x_s	
3)	y_s	y_p	x_s	x_p	
4)	y_p	y_s	x_p	x_s	
5)	y_s	y_p	x_p	x_s	
6)	y_p	y_s	x_s	x_p	

Table 2: Interchange alternatives

y_s is not a spanning dimension in the DVP for any of the basic sets, so the lower bound storage requirements will stay the same as for the unordered situation. The upper bound requirements are reduced however, by a factor 32 everywhere (except for sad(6)) since y_s can be removed from every Dependency Part.

	no ordering	y_p outermost
sad(0)	1/1024	1/1024
sad(1)	1/1024	1/1024
sad(2)	1/1024	1024/1024
sad(3)	1/15360	1/1024
sad(4)	1/15360	1/2048
sad(5)	1/14336	1025/14336
sad(6)	1/1	1/1
Simultaneously alive	1/45056	1025/17408
curr(0)	1/256	1/256
Storage requirement	2/45312	1026/17664

Table 3: Estimated lower/upper bounds for storage requirement of sad[][][][] and curr[][] arrays

The designer can get good hints for the rest of the ordering through an inspection of the DVPs. Non-spanning dimensions reduce the upper bound without altering the lower bound if they are placed outermost, while spanning dimensions with similar reasoning should be placed innermost. In this case the DVPs show that the smallest dependencies can be achieved when y_p and x_p are ordered as the innermost loops, as in interchange alternative 1). Table 4 shows the change of estimated upper and lower bounds as the execution order is fixed. As the dimensions are specified, the upper and lower bounds for the DVPs gradually converge and finally come together when the ordering is fully specified. The difference between the upper and lower bounds for simultaneously alive basic sets even when the ordering is fully fixed, is due to the loop structure of the data-flow graph. A closer inspection of the code reveals that the lower bound is indeed reachable.

basic set	a)	b)	c)	d)
sad(0)	1/32	1/1	1/1	1/1
sad(1)	1/32	1/1	1/1	1/1
sad(2)	1/32	1/1	1/1	1/1
sad(3)	1/480	1/15	1/1	1/1
sad(4)	1/480	1/15	1/2	1/2
sad(5)	1/448	1/14	1/1	1/1
sad(6)	1/1	1/1	1/1	1/1
Simul. alive	1/1408	1/44	1/4	1/4
curr(0)	256/256	256/256	256/256	256/256
Storage req.	257/1664	257/300	257/260	257/260

Table 4: Estimated lower/upper bounds for storage requirement of sad[][][][] array with stepwise fixation of execution ordering; a) y_s outermost, b) x_s 2nd outermost, c) y_p 3rd outermost, d) x_p 4th outermost

5. CONCLUSIONS

We have presented a novel technique for estimating storage requirements for algorithms with partly fixed execution ordering. The methodology can be used during the design trajectory for a large set of data dominated applications in many typical hardware/software codesign domains such as multi-media and telecom applications. Upper and lower bounds on the storage requirement are presented to the designer, who can then use them effectively in early system trade-offs. As more and more of the execution ordering is specified, our upper and lower bounds have the very desirable property that they converge. Using a real life MPEG-4 application, we have demonstrated how the methodology can be used during the early system design trajectory.

6. REFERENCES

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

[7] Tseng, C-J., and Siewiorek, D.P. "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

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

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