Abstract -- This paper describes a new memory estimation technique for DSP applications written in an applicative language. Since no concept of storage is present in an applicative language, the compiler has to derive the memory needs. For each array, the number of storage locations is estimated by modelling both, the dependencies and the sequence of execution. For this, an array-specific production time axis is created and the current production date is compared with the production date of the consumed signal. It is formulated as an integer linear problem.

I. INTRODUCTION.

This paper describes a new memory estimation technique for DSP applications written in the applicative language SILAGE, [1][2]. SILAGE is a data flow language, which is a natural way to describe the flow of computations in DSP applications. Modern DSP applications, such as image and video applications, operate not only on scalar signals but also on more-dimensional arrays of signals. Moreover, for real-time sampled data systems, references to previous sample periods need to be easily specified. Arrays of signals, and a natural extension of it, delayed versions of arrays, are supported by SILAGE.

Since no concept of storage is present in an applicative language, the compiler has to derive the necessary memory locations. For scalar signals, single assignment checking, data dependency analysis and allocation of a storage location is based on the name. A memory location is occupied from the production of the signal until the last consumption. For arrays of signals, the indices of the array have to be evaluated. In many cases, the number of memory locations does not have to be the size of the array (i.e. the multiplication of all its dimensions). It is therefore necessary to estimate the number of signals from the array which are alive at the same time.

A new technique has been developed to estimate the storage needs for arrays of signals. For each array, the number of storage locations is computed by setting up an array-specific production time axis and comparing the current production date with the production date of the consumed value. This can be formulated as an integer linear problem. The model for the representation of arrays and loops is presented in section 2.

Before presenting the method of estimation in section 3, two different, simpler methods will be explained first. The first method consists of counting the number of integer points in the intersection of a production and a consumption node. This can give very large over estimations, and even unbounded estimations for arrays with delays, since it does not take into account the sequence of execution. A second method, which does take the sequence of execution into account, is based on the construction of a “distance vector”, i.e. the distance in loop iterations between a production and a consumption node. It is however restricted to “perfectly nested loops”, i.e. production and consumption are located in the same loop body, [3]. The proposed technique models both, the dependencies between productions and consumptions and the sequence of execution. Its complexity is independent of the size of the arrays. It is more accurate than counting the number of integer points and it can be considered as a generalization of the “distance vectors”, since it is not restricted to consumptions and productions occurring in common loops.

The proposed method gives a more global vision on the problem. This will be explained in section 4, together with results of the proposed method. In section 5, conclusions are formulated and future work is indicated.

The estimation of memory sizes is related to the problem of data dependency analysis, which has been studied in parallelizing and vectorizing compilers to improve the execution performance of nested loops on parallel machines [3][4][5]. It allows to check the validity of loop transformations, such as blocking, tiling, interchange, skewing, etc. [5]. We make use of the OMEGA test, which is an ILP solver based on the Fourier-Motzkin elimination method [6]. It is originally developed to solve the integer linear problems occurring in data dependency analysis, [4].

In high level synthesis, memory size estimation has only recently gained attention. In previous SILAGE compilers [1][7], data dependency analysis and memory size estimation are based on symbolic enumeration. It means that all possible
index combinations of indexed arrays are evaluated and enumerated. This technique is robust but makes the problem proportional to the sizes of the arrays, which is unacceptable.

ILP based techniques are applied in [8][9][10]. In [8], the loop hierarchy and sequence of execution as specified by the designer is deliberately ignored and only the pure data flow of the source code is retained. Next, memory size is minimized by constructing a completely new control flow by placing polytopes in a common space and searching for an ordering vector in this space. It can be at the cost of other goals, such as memory bandwidth or a complex control and address arithmetic. In [9], a technique similar to counting the number of integer points (see section III.A) is used to estimate memory size. Only exact estimations are obtained after loop unrolling, which makes the approach proportional to the sizes of the arrays, similar to the symbolic enumeration technique. In video applications, arrays are accessed in a periodical way, i.e. the clock rate to sample rate is tight and the number of clock cycles between two pixel accesses, two line accesses etc. is often fixed. In the stream model of [10] this periodicity is used to model the sequence of execution. Next, an ILP formulation is used to minimize the memory buffers between video streams.

II. SPECIFICATION.

Unique to the techniques proposed here, is that they do model delays on arrays and give accurate results for memory size estimation even in the presence of delays on the arrays. SILAGE is an applicative language: new signals are created by applying operations on other signals. A description of a DSP algorithm in SILAGE is repeated for each new sample or array of samples arriving. Therefore, a SILAGE program describes the operations of the current sample period, similar to a flow graph description. An adaptive LMS filter for echo cancellation is used to illustrate the concepts of SILAGE and the representation of arrays, shown on Fig. 1. The number of taps, NTAPS (set to 128 for the rest of the paper), will not affect the complexity of the method.

To access samples from previous sample periods, the delay operator @ is used. For instance, in Fig. 1, the current coefficient, Coeff[i], is computed from the previous value of the coefficient, Coeff[i]@1 and the (i+1)th delayed version of the input, ine@(i+1). It appears as a one-dimensional array, although time adds a second dimension to the array. In general, time is considered as an extra dimension to the arrays with delays.

An indexed array at the left hand side or right hand side of a statement, is called a production or consumption node respectively. To each node, a set of linear equalities and inequalities is associated which describes the subscripts of the node as a function of the loop indices and the time, the latter only if delayed consumptions occur in the description. s is the subscript vector, i is the loop iterator vector, A, B, c, d, are matrices (capital letters) and vectors (small letters) of integer values:

\[ s = A \times i + c, \quad LB \leq i \leq UB \]

The second equation, describing the loop bounds, will be written as: \( B \times i + d \geq 0 \).

E.g., the representation of the consumption node ine@(i+1) of Fig. 1 is:

\[
\begin{bmatrix}
1 & -1
\end{bmatrix}
\begin{bmatrix}
\text{ine@i} \\
\text{ine@(i+1)}
\end{bmatrix}
\]

The C and P subscripts on the index variables, refer to the index values during consumption or production respectively. Loop bounds do not need to be constant and can be function of surrounding loop variables, as long as the dependency can be expressed by a linear expression. Remark that time runs from 0 to \( 127 \).

This representation is first used to perform a number of data flow analysis checks, such as a single assignment check, a production-consumption check and the creation of data dependency edges between production and consumption nodes. The details of these techniques are reported in [11].

III. MEMORY SIZE ESTIMATION FOR ARRAYS.

First, we will present two methods which only give accurate results under certain control flow assumptions. The first consists of counting the number of integer points in the intersection of a production and a consumption node. The second consists of counting the number of iterations between a production and the corresponding consumption when both the production and the consumption appear in the same loop body. Thirdly, we present a new method, which is at least as accurate as both methods, gives a more global vision on the
problem and puts no restrictions on the location of the production and consumption nodes.

For all three techniques presented, the following assumptions are made regarding the hierarchy and the global control flow. Loops are executed in the sequence as specified in the source code. For nested loops, it means that iterations of more inner loops run faster. For a sequence of loops, it is assumed that the first loop is finished before the second is started. Within a loop body, no assumption is made on the sequence of execution of the statements in the loop body. If after this high level estimation phase, the scheduler detects that no dependencies exist between two loops, it can still decide to execute them in parallel. With this approach, the designer gets to see the effect of a particular control flow. In a similar way, the designer will get feedback on the memory bandwidth, possible memory types, in-place storage, etc. (not discussed in this paper). Loop transformations or other control flow transformations can then be applied to improve any of these features, depending on the optimization goal (area, speed, power).

A. Method 1: Counting the number of integer points.

One way to estimate the memory size, is by counting the number of integer points in the intersection of a production and a consumption node. It can be formulated as the following integer linear problem: Is there a solution for the subscripts corresponding to the increment of each iterator: 

\[ s = A_p \times i_p + c_p = A_c \times i_c + c_c \quad \text{with} \quad B_p \times i_p + d_p \geq 0 \quad \text{and} \quad B_c \times i_c + d_c \geq 0. \] (2)

This ILP system is solved for the subscript vector \( s \), with the OMEGA test, [4]. If no solution exists, there is no dependency and no storage locations have to be reserved from the production to the consumption node. Exact counting the number of integer points is a numerically hard problem and is only solvable in polynomial time for limited number of dimensions (4D as is shown in [12]). To get an estimation of the memory size, the extreme values of each of the subscripts are queried and multiplied.

\[ \text{MemSize} = \prod_{s} (\max_s - \min_s + 1) \] (3)

with \( \max_s, \min_s \), the extreme values of subscript \( s \). This mapping results in a counting of the number of integer points in the rectangular box around the solution set.

To illustrate the method, consider the overlap between the consumption node \( \text{est}[i] \) and the production node \( \text{est}[i+1] \). For the production node:

\[ [s] = [i_p] \cdot [1] \times [i_c] + [0] \geq 0 \]

For the consumption node:

\[ [s] = [i_c] \cdot [1] \times [i_c] + [0] \geq 0 \]

The intersection is described by: \( 127 \geq s \geq 1 \) and its size is 127, although manual execution shows that only one memory location is needed to store the array \( \text{est}[i] \).

This method gives accurate results for a control flow where all productions of the production node have occurred before the consumptions of the consumption node start and if no unrelated productions occur in between. The latter will be explained in section 4. It will result in a huge over estimation if the productions and the consumptions are intermixed. This occurs e.g. if the production and consumption occur in the same loop body, as is visible in the above example. A more severe restriction is however, that this method will result in an unbounded estimation (\( \infty \)) in case of delayed arrays, since time runs to \( \infty \). This will be the case for each of the other arrays of Fig. 1.

B. Method 2: Counting the number of iterations.

A second method, which is derived from the concept of “perfectly nested loops”, [13][5], consists of counting the number iterations between a production and the corresponding consumption when both the production and the consumption appear in the same loop body. This can be formulated by the following integer linear problem:

\[ \Delta T = b_s \times \Delta i, \quad \Delta i = i_c - i_p \]

\[ A_c \times i_c + c_c = A_p \times i_p + c_p \]

\[ B_p \times i_p + d_p \geq 0 \quad B_c \times i_c + d_c \geq 0 \] (4)

The last three equations again describe the constraints for a dependency between the production and the consumption node. \( \Delta i \) is defined as the distance vector. It stores the difference in number of iterations between the consumption and the production node, [13]. It assumes that loops are normalized, i.e. have a step of 1. To obtain an estimation of the total number of iterations, a projection on the iteration time axis is made. This is described by the first equation. \( \Delta T \) is a scalar, returning the absolute number of iterations of the innermost loop between the production and the consumption. \( b_s \) is the bound-size vector. It describes the number of iterations corresponding to the increment of each iterator: \( i_j \) to \( i_{N_j} \).

\[ b_s^T = [\text{one}_{i_1} \, \text{one}_{i_2} \, \ldots \, \text{one}_{i_N}] \] (5)

This bound-size vector describes the sequential execution of the loops. Assume the loops are executed from \( i_j \) to \( i_{N_j} \). Then “one \( i_j \)” is 1 for the innermost loop and for the other loops:

\[ \text{one}_{i_j} = \prod_{k=j+1}^{N} (UB_k - LB_k + 1) \quad \text{for} \ j = 1 \ldots N-1 \] (6)

The number of iterations between the production node \( \text{Coeff}[i] \) and the consumption node \( \text{Coeff}[i] \otimes 1 \), is
solved by the following problem:

\[ [\Delta T] = [128 \ 1] \times [\Delta] \]

\[ [\Delta] = \begin{bmatrix} t_c - t_p \\ i_c \\ i_p \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 0 & [i_c] + [-1] \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & t_p \\ 0 & 1 & i_p \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix} \geq 0 \]

This results in a difference \( \Delta T \) of 128. Manual evaluation of the code indeed shows that for the storage of the array \( \text{Coeff} \), 128 storage locations are needed.

Since there is a one to one correspondence between the number of iterations and the number of productions, counting the number of iterations gives an accurate estimation of the number of memory locations between the current consumption and the corresponding production. This will be the start time the node occupies a memory location. In a second step, during consumption, the number of iterations gives an accurate estimation of the number of storage locations needed for a node, the current production time is compared to the production time of a node for the array under investigation. This results in a difference \( \Delta T \) of 128. Manual evaluation of the code indeed shows that for the storage of the array \( \text{Coeff} \), 128 storage locations are needed.

Since there is a one to one correspondence between the number of iterations and the number of productions, counting the number of iterations gives an accurate estimation of the number of memory locations between the current consumption and the corresponding production some iterations ago. Since \( \Delta T \) is not defined outside the common loop body, this method is restricted to productions and consumptions which occur in the same loop body.

C. Method 3: Comparison on the production time axis.

The usage of the distance vector gives already a good model to represent the sequence of execution. We want to extend it however over the loop boundaries to cover the whole flow graph. Therefore, in a first step, a production time axis will be constructed for the array under investigation, which models the sequence of execution. With this information, a production time in the form of an integer linear expression, can be associated with each node. This will be the start time the node occupies a memory location. In a second step, during consumption of a node, the current production time is compared to the production time of the consumed node. This difference indicates how many values in the array have been produced and stored since then. The maximum difference over all consumptions gives the maximum number of storage locations needed for the array.

C. 1 Setting up the production time axis.

The production time axis of an array models the relative production order of each of the individual signals in the array. For this, the flow graph is traversed hierarchically and sequence numbers, called \( \text{StartTime} \), and production sizes, called \( \text{NumberOfProd} \), are associated with the graphs. These numbers allow to create an integer equation which, when evaluated for a particular index combination, results in a unique production time number. This is illustrated by the example on Fig. 2.

The recursive pseudo-code of Fig. 5 computes \( \text{StartTime} \)’s and \( \text{NumberOfProd} \) for all graphs. At each level in the hierarchy, start times start again from 0, i.e. relative start times are computed. \( \text{NumberOfProd} \), associated with a graph, stores the number of productions corresponding to one increment of the index of that graph. The main difference with an iteration time axis, is that if a subgraph does not contain production nodes, production time will not increase. It will result in a \( \text{NumberOfProd} \) which is 0 for the subgraph, and the \( \text{NextStart} \) will be identical to the current start time. Two smaller routines are used in this algorithm: \( \text{ProductionNodes}() \) counts the number of statements in a leaf graph which contain a production node for the array under investigation. \( \text{SizeOfGraph()} \) computes the number of iterations corresponding to the graph:

\[
\text{SizeOfGraph} = \begin{cases} 1 & \text{for cluster} \\ UB_i - LB_i + 1 & \text{for loop} \end{cases}
\]

C. 2 Creation of Production Time Expression.

Once this production time axis is constructed, the production time of a production node can be computed by adding all start times of its enclosing loops, together with the offset of the loop indices compared to their lower bound. It results in the following equation:

\[
\text{FinalDate} = \text{CreateProdTimeAxis}(\text{TopGraph}, 0); \\
\text{int CreateProdTimeAxis}(\text{Graph}, \text{Start}) \\
\text{Graph}->\text{StartTime} = \text{Start}; \\
\text{if} (\text{leaf graph}) \\
\#PR = \text{ProductionNodes}(\text{Graph}); \\
\text{Graph}->\text{NumberOfProd} = \#PR; \\
\text{return}(\text{Start} + \#PR * \text{SizeOfGraph}(\text{Graph})); \\
\text{else} \\
\text{NextStart} = 0; \\
\text{for each SubGraph inside Graph at this level} \\
\text{NextStart} = \text{CreateProdTimeAxis}(\text{SubGraph}, \text{NextStart}); \\
\#PR = \text{NextStart}; \\
\text{Graph}->\text{NumberOfProd} = \#PR; \\
\text{return}(\text{Start} + \#PR * \text{SizeOfGraph}(\text{Graph}));
\]

Fig. 2: Construction of the production time axis for array \( a[][] \).

Fig. 3: PseudoCode to construct the production time axis
\[ PT = \sum_{i} (Start_i + (i - LB_i) \times PR_i) \] (8)

An increment of the surrounding loops will result in a number of extra signals given by \( NumberOfProd \). The outer most graph will be the time loop. An increment of the time loop will result in a number of signals given by \( FinalDate \). Evaluating this expression for a particular index combination will result in an integer number which corresponds to the production date.

In computing \( SizeOfGraph() \) and \( NumberOfProd() \), varying loop bounds are replaced by their upper limits to obtain integer values. As a consequence, the production time can be expressed as an integer equation in the loop indices surrounding it. If index expressions would be allowed in the expressions for \( NumberOfProd \) or \( SizeOfGraph \), non linear, i.e. quadratic, type of expressions, could be obtained for the production time, which prohibits solving the problem with an ILP technique. Eq. (8) is thus of the form:

\[ PT = bs \times i + d \] (9)

C.3. Dependency analysis and distance computation.

With this information, the number of memory locations for an array can be obtained by comparing the current production date with the production date of the consumed value, which can be formulated as follows:

\[
\begin{align*}
\Delta T &= PT_C - PT_P \\
PT_C &= b_s \times i_C + d_C \\
PT_P &= b_s \times i_P + d_P \\
A_C \times i_C + c_C &= A_p \times i_p + c_p \\
B_P \times i_P + d_P &\geq 0 \\
B_C \times i_C + d_C &\geq 0
\end{align*}
\] (10)

Again, the last three equations formulate the constraints for dependency between the production and the consumption node. \( PT_P \) is the production time of the production node. \( PT_C \) is the current production date. \( \Delta T \) describes the difference between the two dates and thus the number of memory locations needed between the consumption and the production date. If the solution for \( \Delta T \) varies, the extreme values are evaluated to obtain the memory locations. Remark that if negative values for \( \Delta T \) are obtained, this indicates a consumption before a production, and thus code which is not executable in the given sequence.

E.g., the number of storage locations between the consumption of \( a[i-1][j] \) and the production node \( a[0][k] \) in Fig. 2, is \( M+1 \). It is obtained by solving:

\[ \begin{align*}
\Delta T &= PT_C - PT_P \\
PT_C &= (M + 1) \times i_C + [-1]
\end{align*} \]

\[ \begin{align*}
[PT_C] &= \begin{bmatrix} 1 \end{bmatrix} \times \begin{bmatrix} i_C \end{bmatrix} + [-1]
\end{align*} \]

\[ \begin{align*}
[PT_P] &= \begin{bmatrix} 0 \end{bmatrix} \times \begin{bmatrix} i_P \end{bmatrix} + 0
\end{align*} \]

\[ \begin{align*}
0 \times \begin{bmatrix} i_P \end{bmatrix} + 0 &= \begin{bmatrix} 0 \end{bmatrix} \times \begin{bmatrix} i_C \end{bmatrix} + [-1] \\
1 \times \begin{bmatrix} i_C \end{bmatrix} + [-1] &\geq 0, \begin{bmatrix} 0 \end{bmatrix} \times \begin{bmatrix} i_C \end{bmatrix} + N \geq 0
\end{align*} \]

C.4. The global algorithm.

For each array, a production time axis has to be created, and each consumption node has to be compared to each production node of a particular array. The maximum of the \( \Delta T \) values in each comparison is the total amount of memory locations needed.

IV. DISCUSSION.

To show that this approach gives a more global vision on the number of memory locations, the productions and consumptions of the example of Fig. 2 are visualized on Fig. 4. The production nodes are shown with the production dates, the consumption nodes with the current production dates and the distances each computed with Eq. (10). The number of memory locations needed between the consumption of \( a[i-1][j] \) and the production node \( a[0][k] \), results in an estimation of \( M+1 \) storage locations. On the contrary, counting the number of integer points in the intersection, indicates for this example \( M \) storage locations. Between the production of \( a[0][k] \) in graph_k and its consumption in \( a[i-1][j] \) in graph_j, lies the production of \( a[i][0] \). Since production time has increased when creating \( a[i][0] \), memory locations are reserved for it on the production time axis. Taking the difference between the production dates will thus take memory locations for this unrelated production into account.

The same problem occurs when counting the number of iterations. One will obtain \( M_j \)-loop iterations between the production \( a[i][j] \) and the consumption \( a[i-1][j] \), while the method presented here will result in \( M+1 \) memory loca-

<table>
<thead>
<tr>
<th>Productions:</th>
<th>Consumptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Prod. Date</td>
<td>Node Current Prod. Date</td>
</tr>
<tr>
<td>a[0][k]</td>
<td>k-1</td>
</tr>
<tr>
<td>a[i][0]</td>
<td>(M+1,i-1)</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>(M+1,i+j-1)</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>(M+1,i+j+1)</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>(M+1,i+1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons. Node Depends on:</th>
<th>( \Delta T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i][j]</td>
<td>a[i][0]</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>a[i][j]</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>a[0][k]</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>a[i][j]</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>a[i][j]</td>
</tr>
<tr>
<td>a[i][j]</td>
<td>a[i][j]</td>
</tr>
</tbody>
</table>

Max: \( M+1 \)

Fig. 4: Production dates and dependencies for the example of Fig. 2.
In this paper, a model is proposed to represent arrays in the applicative language SILAGE. Time delayed versions of arrays are easily specified in this model, by adding an extra dimension to the arrays. Next, a technique is presented to estimate the necessary memory locations for each array occurring in the SILAGE description. It combines a model to represent the sequence of execution, called the production time axis, and a model to represent the dependencies. By comparing the current production date and the production date of the consumed value, an estimation of the memory size is obtained. This method gives a more global vision on the problem than two other methods, which consist of counting the number of integer points, and counting the number of iterations, respectively.

Since we consider estimation as a fundamental part of any design exploration environment [14], future work consists of giving the designer even more feedback, not only on the number of memory locations, but also on where the memory locations are needed, to allow e.g. in-place storage of different arrays in the same memory. Furthermore, we are combining this memory size estimation technique with a memory bandwidth estimation technique and a memory type selection technique. With this feedback information, the designer can make more intelligent decisions on where to optimize for area, speed or power.

TABLE I: COMPLEXITY AND RUN TIMES FOR SEVERAL APPLICATIONS.

<table>
<thead>
<tr>
<th>Application</th>
<th>A</th>
<th>P/C</th>
<th>SL</th>
<th>ILP</th>
<th>%</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lms</td>
<td>4</td>
<td>14</td>
<td>23</td>
<td>12</td>
<td>0.25</td>
<td>30</td>
</tr>
<tr>
<td>TSVQ</td>
<td>11</td>
<td>40</td>
<td>72</td>
<td>36</td>
<td>0.28</td>
<td>120</td>
</tr>
<tr>
<td>echo</td>
<td>6</td>
<td>29</td>
<td>96</td>
<td>42</td>
<td>0.50</td>
<td>100</td>
</tr>
<tr>
<td>complex lms</td>
<td>24</td>
<td>74</td>
<td>97</td>
<td>60</td>
<td>0.73</td>
<td>220</td>
</tr>
<tr>
<td>histogram</td>
<td>10</td>
<td>52</td>
<td>113</td>
<td>56</td>
<td>0.30</td>
<td>150</td>
</tr>
<tr>
<td>vocoder</td>
<td>17</td>
<td>89</td>
<td>122</td>
<td>128</td>
<td>0.34</td>
<td>340</td>
</tr>
<tr>
<td>codec</td>
<td>30</td>
<td>171</td>
<td>591</td>
<td>189</td>
<td>0.51</td>
<td>500</td>
</tr>
</tbody>
</table>

A = number of arrays, P/C = number of production&consumption nodes
SL = number of SILAGE lines, ILP = number of ILP problems
% = average number of memory locations/arraysize for all arrays

V. CONCLUSIONS.

REFERENCES.