

SPECIFICATION AND SUPPORT FOR MULTI-DIMENSIONAL DSP IN THE SILAGE LANGUAGE.

Ingrid M. Verbauwhede¹, Chris J. Scheers², Jan M. Rabaey¹,

¹EECS Department, University of California at Berkeley, Cory Hall, Berkeley CA 94720, U.S.A.

²Zycad Corporation, Fremont, CA, U.S.A.

ABSTRACT

Data flow languages are a natural way to describe the flow of computations in a DSP application. The SILAGE language has been developed for this purpose. It contains also more-dimensional arrays of signals and a natural extension of it, delayed versions of arrays, e.g. to represent previous frames in video applications.

This paper describes new data flow analysis techniques, to support multi-dimensional arrays. It checks single assignment of arrays, checks if for each consumption of an indexed signal, there is a production, and it will create data dependencies between productions and consumptions. These problems are formulated as integer linear programming problems. This formulation is independent of the number of signals in the arrays. Results show very fast running times (< 1s) for problems more than hundred nodes.

1. INTRODUCTION

Data flow languages are a convenient way to capture the flow of computations in a DSP application. The SILAGE language, [1][2], has been developed for this purpose and contains a number of special constructs, e.g. the sample delay, that are commonly found in DSP applications. State of the art applications, such as image and video applications, operate not only on single scalar signals, but also on more-dimensional arrays of signals. SILAGE supports operations on arrays and extends in a natural way to delayed versions of arrays.

For scalar signals, single assignment checking and creating data dependencies between operations is based on their name. Each scalar has its own name and, when ordering the statements, a definition of a scalar has to occur before a consumption of a scalar. For multi-dimensional arrays of signals however, the indices of the array have to be evaluated. This paper describes new data flow analysis techniques, which explicitly support multi-dimensional arrays. First, it checks the single assignment of arrays. Secondly, it checks whether for each consumption of an indexed signal, there is a production. Thirdly, it

will create data dependencies between productions and consumptions of signals. Each of these problems can be formulated as an integer linear programming problem, which are solved with the Omega test, [4].

Previous SILAGE parsers, such as the public domain S2C simulator used e.g. in [5], make use of symbolic enumeration techniques. This has the disadvantage that the enumeration tables grow with the number of signals in the array and become unacceptably large for e.g. video frames with delays. The techniques proposed here are independent of the sizes of the arrays and are based on an ILP formulation. In high level synthesis, ILP formulations are being used for memory management and for obtaining a global control flow, i.e. a global ordering of loops and the loop hierarchy, in a dataflow description, [6][7][8].

ILP formulations for data dependency analysis have been used for conventional procedural languages, [4][9][10]. We have adapted the techniques for the typical problems occurring in data flow languages, such as the single assignment check. Moreover, since a DSP program is assumed to run continuously on streams of data [2], the concept of time adds an extra dimension to the arrays. This time concept can be smoothly incorporated in the representation model of arrays.

In section 2, the representation of arrays and sample delays in SILAGE is introduced. In section 3, the different data dependency checks are discussed. In section 4, some results are presented. Section 5 contains the conclusion and future work.

2. SPECIFICATION.

The description of arrays of signals in SILAGE will be illustrated by a small example, taken from a 128-taps LMS filter from an echo cancellation example. For clarity, initializations and types are omitted. For a complete description of SILAGE, we refer to [3]. The kernel of the filter is the computation of the estimated error and the updating of the coefficients as shown in Figure 1.

SILAGE is an applicative language: new signals are created by applying operations on other signals. DSP algorithms are repeated for each new sample arriving.

```

est[0] = 0;
(i : 0 .. NTAPS - 1) ::
begin
  Coeff[i]@1= 0; /* initialization */
  ine@(i+1)= 0; /* initialization */
  est[i+1]= est[i] + ine@i * Coeff[i];
  Coeff[i]= Coeff[i]@1+(alpha * ine@(i+1))
end;
oute = backe - est[NTAPS];

```

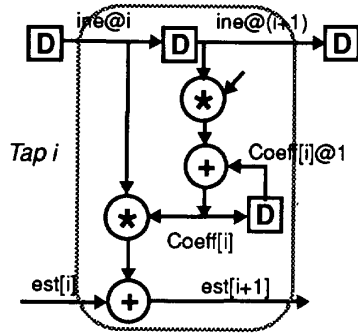


Figure 1: SILAGE description and corresponding filter block diagram.

Therefore, a SILAGE program describes the operations of the current sample period, similar to a signal flow graph description. To access samples from previous sample periods, the delay operator, @, is introduced. For instance, in Figure 1, the current coefficient, $Coeff[i]$, is computed from the previous value of the coefficient, $Coeff[i]@1$ and the $(i+1)$ -delayed version of the input, $ine@(i+1)$. Expressions in SILAGE indicate infinite streams of data. As such, time can be considered as an extra dimension to the arrays.

3. SUPPORT FOR MULTI-DIMENSIONAL ARRAYS

An occurrence of an indexed array at the left hand side or the right hand side of a statement, is called a production or consumption node respectively. With each node, a set of linear equalities and inequalities is associated, which defines the subscripts of the node as a function of the loop indices and the time, 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:

$$s = A \times i + c \text{ with } LB \leq i \leq UB$$

The latter part can be rewritten as: $B \times i + d \geq 0$

c and p refer to the index values during consumption or production, $s1, s2$ refer to the first, second subscript, t_p, t_c refer to the time loop index and $i_{c1}, i_{c2}, i_{p1}, i_{p2}$, etc. to loop indices. Time adds an extra dimension to the arrays, therefore $Coeff[i]$ has two dimensions:

$$\begin{bmatrix} s1 \\ s2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} t_p \\ t_c \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} t_p \\ t_c \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 127 \end{bmatrix} \geq 0$$

The representation of the consumption node

$$ine@(i+1) \text{ is: } \begin{bmatrix} s1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix} \times \begin{bmatrix} t_c \\ i_c \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} t_c \\ i_c \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 127 \end{bmatrix} \geq 0$$

Remark that time runs from 0 to ∞ . Therefore, there is only a lower bound on the time index. In general, the subscript vector is a linear function of the loop indices surrounding the node. The loop boundaries are described by a set of linear inequalities. Loop bounds do not need to be constant. They can be linear expressions of surrounding loop indices.

3.1 Single assignment check:

If several production nodes define parts of the same array, we have to check if those definitions do not overlap. This is equivalent to solving the following problem: *Is there an integer solution for the subscripts for the following set of linear equalities and inequalities, which describes the intersection of two production nodes?*

$$s = A_{p1} \times i_{p1} = A_{p2} \times i_{p2} \text{ with } B_{p1} \times i_{p1} + d_{p1} \geq 0 \text{ and } B_{p2} \times i_{p2} + d_{p2} \geq 0$$

with s = vector of subscripts, i_{p1}, i_{p2} = vector of loop indices surrounding production node 1 and node 2 respectively. If a solution exists, single assignment is violated. This pairs-wise comparison is needed between all production nodes of the same array.

3.2 Production-consumption check:

All integer points in a consumption node need to be covered by one or more production nodes. Simply taking the intersection of the production and the consumption node, does not solve the problem. We need to know whether integer points of the consumption node lie outside the production node. Therefore, we check if the system described by the consumption node augmented with one of the inequality constraints of the production node inverted, has a solution. If so, this describes the part of the consumption node, which lies outside the production node. This new system has to be covered by the remaining production nodes. This process is repeated recursively until the node is covered or no production nodes are left. The latter indicates a violation. The pseudo-code for this production-consumption check is given in Figure 2. The routine *ILP-Solve* tries to find an integer solution for the ILP problem

```

CheckProductionsConsumptions(Graph)
for each array,
  for each ConsNode
    IntersectWithComplement
      (ConsNode, ProdNodeList);

IntersectWithComplement(ConsNode, ProdNodeList)
if (ProdNodeList empty)
  ERROR: consumption without production;
else
  take first Prodnode from the list;
  for each constraint of ProdNode
    Problem = ConsNode + inverted constraint;
    Solution = ILPSolve(Problem);
    if (Solution)
      IntersectWithComplement
        (Solution, ProdNodeList->Next);
  else
    continue;
endfor;
return OK;

```

Figure 2: Pseudo code for the production consumption check.

with the constraint added. Remark that, if there is no solution for each of the production constraints inverted, then the consumption node is completely covered by the production node. If however, a solution is found for one of the constraints, only the following conclusion can be made about the coverage: the production node can cover the consumption node partly or not at all.

For example, the consumption node $\text{Coeff}[i]@1$ is covered by two production nodes. The representation for the consumption node is the following:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 127 \end{bmatrix} \geq 0$$

Remark that a reduced representation as a function of the subscripts is used, i.e. all loop indices and redundant constraints are removed. The production node, $\text{Coeff}[i]$ is represented as follows:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 127 \end{bmatrix} \geq 0$$

Only adding the first inequality of the production node in its inverted form, to the consumption node representation will result in an ILP problem with a solution. The augmented and the simplified representation are as follows:

$$\text{problem before reduction: } \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix} \times \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 127 \\ -1 \end{bmatrix} \geq 0$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \times \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 0$$

$$\text{after reduction: } \begin{bmatrix} 1 \\ -1 \end{bmatrix} \times \begin{bmatrix} s_2 \\ 127 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \geq 0$$

Since time starts at $t=0$, this part of the consumption node will be covered by the initialization node, $\text{Coeff}[i]@@1$, which is the production of $\text{Coeff}[]$ at time $t=-1$.

3.3 Creation of data dependencies:

The SILAGE description is input for several high level synthesis environments, such as the HYPER and the CATHEDRAL environments [11]. To establish ordering between operations in a dataflow representation, data dependency edges are created.

The dependencies between productions and consumptions can be created by checking the intersection between a production and a consumption node, in a similar way as the checking of the single assignment: *Is there an integer solution for the subscripts for the following set of linear equalities and inequalities, which describes the intersection of a production node and a consumption node?*

$$s = A_p \times i_p = A_c \times i_c \text{ with } B_p \times i_p + d_p \geq 0 \text{ and } B_c \times i_c + d_c \geq 0$$

If there is a solution, a dependency exists between the production and the consumption node. For each array a pairs-wise comparison between each production node and each consumption node is needed. For instance, a dependency exists from the production node $\text{est}[0_p]$ to the consumption node $\text{est}[i_c]$, but also from the production node $\text{est}[i_p+1]$ to the consumption node $\text{est}[i_c]$.

A legal ordering of the operations, which makes the code executable, is obtained if productions occur before consumptions. However, in general, a production or consumption node represents a set of signals. Not all signals in this set have to be produced before the first of the consumptions can start. Requiring that all productions occur before consumptions can result in cyclic dependency requirements, as e.g. the above dependency between production $\text{est}[i_p+1]$ and consumption $\text{est}[i_c]$, since the production of the next $\text{est}[i_p+1]$ depends through some operations on the current $\text{est}[i_c]$. Moreover, this requirement can lead to unbounded memory requirement if arrays with delays are involved. Therefore, a notion of dependencies within the sets and a relative order of execution has to be added to this data dependency checking step. The details of this problem formulation and solution are reported in [8].

Table 1: Results:

Application	128T lms		TSVQ		histogram		vocoder		codec	
	#P	time	#P	time	#P	time	#P	time	#P	time
#Arrays	4		11		10		17		30	
#Prod./Cons. nodes	14		40		43		90		172	
Single Assign.	11	10	36	40	26	30	72	70	98	90
Prod. - Cons.	31	10	126	40	120	30	375	80	326	110
Dependency creat.	12	20	36	80	46	90	131	270	218	450

3.4 Solution Approach:

The techniques proposed here are independent of the sizes of the arrays. Each of the problems above is formulated as an integer linear programming problem, which is solved by the Omega test, [4]. It makes use of the Fourier-Motzkin elimination method to solve a system of linear inequalities. Its complexity grows worst case exponential with the size of the problem. The problem is however only proportional to the number of subscripts and loop indices, which is in most practical cases very small. Our test results agree with the conclusion in [4], that the Omega test runs in effective polynomial time.

Traditionally, this test is used in parallelizing Fortran compilers (and imperative languages in general). The major difference is that we have it applied for very specific checks which are unique to a dataflow language such as SILAGE.

4. RESULTS.

The above data dependency techniques are incorporated in the front-end SILAGE parser of the HYPER synthesis environment [11]. The implementation is done in C and tests are run on a SUN Sparc2 workstation. The applications listed in the table 1, include a 128-taps LMS filter for echo cancellation, a tree search vector quantizer (TSVQ) with a code book size of 256 words, a probability density function computation which takes 4096 samples (histogram), a complete Linear Predictive Coding vocoder (vocoder) and an encoder/decoder for a digital consumer audio application. The table shows the number of arrays in the SILAGE description, the total number of production and consumption nodes in the description, the number of problems to be solved by the OMEGA test and the time in ms.

5. CONCLUSIONS

Data dependency analysis checks for arrays of signals can not simply be based on the name of the array. Index evaluation is needed for this purpose. In this paper, arrays of signals are represented by a set of linear equalities and inequalities. The continuous stream of data, which is particular to DSP applications can be modeled as an extra

dimension to the arrays. Data dependency analysis checks have been presented, which give exact information on single assignment violations, production consumption violations and which can be used for data dependency creation. Current work consists of using the information obtained from these checks in other high level synthesis tasks, such as the estimation of memory sizes.

6. REFERENCES

- [1] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, H. De Man, "DSP Specification using the SILAGE language," *IEEE International Conference on Acoustics Speech and Signal Processing*, pp. 1057-1060, 1990.
- [2] P. Hilfinger, J. Rabaey, "DSP Specification Using the SILAGE Language," Chapter 15 in R.W. Brodersen (ed.), *Anatomy of a Silicon Compiler*, Kluwer Ac. Publ., 1992.
- [3] "Silage User's and Reference Manual," Prepared by Mentor Graphics/EDC, June 1991.
- [4] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependency analysis," *Communications of the ACM*, 8:102-114, Aug. 1992.
- [5] J. Vanhoof, I. Bolsens, H. De Man, "Compiling multi-dimensional data streams into distributed DSP ASIC Memory," *Proceedings of ICCAD'91*, Nov. 1991
- [6] M. van Swaaij, F. Franssen, F. Catthoor, H. De Man, "Modeling data flow and control flow for high level memory management," *Proceedings of EDAC-92*, pp. 8-13, Febr. 1992.
- [7] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. Van Meerbergen, A. van der Werf, "Modelling periodicity by Phideo Streams," *Proc. Workshop on High Level Synthesis*, Dana Point Resort, CA, Nov. 1992.
- [8] I. Verbauwhede, C. Scheers, J. Rabaey, "Memory Estimation for High Level Synthesis," submitted for DAC-94.
- [9] P. Feautrier, "Dataflow analysis of array and scalar references," *Journal of Parallel Programming*, 20(1), 1991.
- [10] M. Wolfe, "Data dependency and program restructuring," *Journal of Supercomputing*, 4, pp. 321-344, 1990.
- [11] J. Rabaey et. al, "Fast Prototyping of Data path intensive Architectures," *IEEE Design and Test*, Vol. 8, No. 2, 1991.