

Exact Evaluation of Memory Size for Multi-dimensional Signal Processing Systems *

Florin Balasa

Francky Catthoor

Hugo De Man[†]

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

Memory cost is typically responsible for up to 80% of the chip and/or board area of most video and image processing system realisations. In this paper, we present a novel technique – founded on data-flow analysis – which allows to address the problem of background memory size evaluation for a given non-procedural algorithm specification. Usually, the number of signal instances is huge, so a new data-flow model grouping scalar signals in so-called basic sets is proposed. The method also incorporates a way to trade-off memory size with computational and controller complexity.

1 Introduction

Application studies in the areas of speech, image and video processing indicate that between 50 and 80% of the area cost in (application-specific) architectures for real-time multi-dimensional signal processing (RMSP) is due to *memory units*, i.e. single or multi-port RAMs, pointer-addressed memories, and register files. This cost would further increase if the data-paths would be optimised without considering the implications on memory. Taking into account that a 128 kbit RAM takes about 115mm^2 in a $1.2\ \mu\text{m}$ CMOS technology, a single buffer already largely dominates the data-path, of which the arithmetic core (without storage units or routing) typically takes 5 to 15 mm^2 . For these reasons, we have opted to optimize first the *high-level storage organisation* for the *multi-dimensional (MD) signals* in our CATHEDRAL script [8]. This high-level memory management (HLMM) stage includes a decision on: the *number and type of (background) memory units*, the *signal-to-memory binding*, and the detailed *internal organisation of the memory units*. Note that this is fully complementary to the traditional high-level synthesis step known as "register allocation/assignment" [5, 1, 4] which deals with individual storage places for scalars, after scheduling. Part of this effort is also needed in the CATHEDRAL context, but this decision on scalar memory management is postponed to our low-level data-path mapping stage [4].

Three main partly conflicting objectives can be identified during HLMM, solvable within different steps: (1) optimizing the memory access by distributing the signals efficiently over a set of background memories and ports (directly related to memory allocation), (2) reducing the *memory size* by in-place storage of signals, and (3) minimising the area overhead of *address generation circuitry*

by optimising the memory access sequences. In front of these steps, we have added a global loop hierarchy optimization technique [8], based on abstract cost measures. As a result, the loop structure can be seen as largely constrained during the subsequent HLMM substeps. The latter have to deal then with a partially restricted search space but are based on more accurate costs. This paper contributes to the memory allocation issue from step (1).

2 Background memory allocation and "quantitative" data-flow analysis

To our knowledge, almost all techniques for dealing with the allocation of storage units employ a *scheduling-directed* view (see e.g. [5, 1] and their references) where the control steps of production/consumption for each individual signal are determined beforehand. This strategy is mainly due to the fact that applications targeted in conventional high-level synthesis contained few loops and even less MD signals. The CDFGs addressed are mainly composed of potentially conditional updates of scalar signals. Hence, the main aim is typically the minimization of the number of registers for storing the scalars. Consequently, the techniques employed to deal with the problem in a global way, could be largely based on ILP formulations. The cost functions are based only on the number of memory cells - which was practically feasible due to the relatively small amount of signals in the test-vehicles.

Both this strategy and these techniques present serious shortcomings for many RMSP applications. First, scheduling *must* proceed HLMM in the high-level synthesis system. This limits the search space available for optimizing the dominating background memory cost. Furthermore, within this view, many examples are untractable because of the huge size of the ILP formulations. An exception to this is PHIDEO [6] where streams are used during the memory allocation, but which is still done after scheduling. A crucial step to overcome these limitations is a method to evaluate the expected area for a given memory before scheduling. This routine can then be used to guide the memory allocation. A precise evaluation of the memory area implies two distinct aspects:

1. assessment of the number and word-length of the storage locations, which determines the total number of memory cells. That is the topic of this paper;

2. determining the number and type (*read, write, or read/write*) of memory ports, which heavily influences the area cost of a single cell. This step is based on a partial scheduling of the read/write operations within the throughput and timing constraints. It will be detailed in a future paper.

*This research was sponsored by the ESPRIT Basic Research Action 3281 (NANA II) project of the E.C.

[†]Professor at the Katholieke Universiteit Leuven

For the assessment of the actual silicon area occupied by the global background memory, the model presented in [7] is employed.

Finding the minimum number of memory locations/registers when an algorithm is described in a fully procedural language, or when the scheduling of the operations was previously accomplished, is solved in e.g. [5] and [4] (for cyclic graphs). Then, the *life times* of variables (signals) can be used in a polynomial "left-edge" type algorithm. Unfortunately, when this information is not directly available, we can prove that the problem becomes NP-complete. A straightforward way of solving the problem under these constraints is by means of an accurate data-flow analysis. Unfortunately, as RMSP algorithms contain a huge amount of signals at the scalar level, a classic data-flow analysis becomes impossible for real-life applications. Finding *all* the dependences at individual signal level in e.g. image processing applications containing millions and even billions of signals is unrealistic. Therefore, a data-flow analysis operating with *groups of signals* rather than *individual signals* is compulsory. This is also the motivation for polyhedral data flow models in array synthesis (see [8] and its references).

Another aspect must be emphasized too. For many of the problems requiring a data-flow analysis, as in parallelism detection, finding the *existence* of dependences is sufficient. However, minimizing the total storage cost requires knowledge about the exact *number* of dependences between the MD signals. Consequently, besides the qualitative aspects, the data-flow analysis must be provided with *quantitative* capabilities. Unfortunately, this can affect seriously (but unavoidably) the computational effort, as it will be seen further.

In order to deal with the memory allocation subtask, a further extension of our polyhedral dependence graph model [8] will be presented in section 3. It is built around the concept of *basic sets* of signals. The assessment of the number of storage locations for algorithms described in a *non-procedural* language is solved by means of a "quantitative" data-flow analysis, presented in section 4. Results obtained so far are substantiated in section 5, followed by the conclusions and our future directions of research in section 6. It is assumed in the sequel that all indices of MD signals, as well as the loop boundaries, are linear functions with integer coefficients of the surrounding loop iterators and that they are manifest (i.e. known at compile time).

3 An analytical partitioning of signals

The set of appearances of an MD signal in the algorithm is referred to as the collection of definition (left-hand side) and operand (right-hand side) domains for that signal [8]. Each domain can be expressed by an affine function over a set of linear inequalities representing a polytope,

$$\{ \mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b} \} \quad (1)$$

where \mathbf{x} is the coordinate vector of the signal, and \mathbf{i} is the vector of loop iterators. \mathbf{T} , \mathbf{A} , and \mathbf{u} , \mathbf{b} are matrices respectively vectors with integer elements.

Once the collections of definition/operand domains are extracted from the RMSP algorithm, a partitioning process into non-overlapping "pieces" - called *basic sets* - is performed for each collection. E.g. a collection of 2 domains results in at most 3 basic sets; 3 overlapping domains result in maximally 7 sets. The basic operation for this analytical partitioning is the intersection of basic sets. Let

$\{ \mathbf{x} = \mathbf{T}_1 \mathbf{i}_1 + \mathbf{u}_1 \mid \mathbf{A}_1 \mathbf{i}_1 \geq \mathbf{b}_1 \}$, $\{ \mathbf{x} = \mathbf{T}_2 \mathbf{i}_2 + \mathbf{u}_2 \mid \mathbf{A}_2 \mathbf{i}_2 \geq \mathbf{b}_2 \}$ be two basic sets, where \mathbf{T}_1 and \mathbf{T}_2 have the same number of rows (as they are derived from the same MD signal). Intersecting the two basic sets means, first of all, solving a linear Diophantine system $\mathbf{T}_1 \mathbf{i}_1 - \mathbf{T}_2 \mathbf{i}_2 = \mathbf{u}_2 - \mathbf{u}_1$ having the elements of \mathbf{i}_1 and \mathbf{i}_2 as unknowns. If the system has no solution, the intersection is empty. Otherwise, let

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \mathbf{i} + \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}$$

be the solution of the Diophantine system. If the set of coalesced constraints has at least one integer solution, than the intersection is a new basic set defined by

$$\mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} \quad \text{where } \mathbf{T} = \mathbf{T}_1 \mathbf{V}_1, \mathbf{u} = \mathbf{T}_1 \mathbf{v}_1 + \mathbf{u}_1$$

$$\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b} \quad \text{with } \mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \mathbf{V}_1 \\ \mathbf{A}_2 \mathbf{V}_2 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 - \mathbf{A}_1 \mathbf{v}_1 \\ \mathbf{b}_2 - \mathbf{A}_2 \mathbf{v}_2 \end{bmatrix}$$

All the known methods for solving a linear Diophantine system are based on bringing the system matrix to the Hermite normal form. They only differ by the modality in which the Hermite normal form is obtained. Our method - using a shorter worst-case sequence of unimodular transformations - is thoroughly described in [2].

The partitioning process - described in the sequel - yields a separate inclusion graph for each of the collections of basic sets. The direct inclusion relation is denoted by " \subset ", while its transitive closure (the existence of a path in the graph) is denoted by " \prec ".

initialize inclusion graph with the def/opd domains as nodes
while new nodes/arcs can be appended to the graph
for each pair of basic sets ($bset_1, bset_2$) in the collection
such that ($bset_1 \not\subset bset_2$) && ($bset_2 \not\subset bset_1$) {
compute $bset = bset_1 \cap bset_2$;
if ($bset \neq \emptyset$)
if ($bset == bset_1$) AddInclusion($bset_1, bset_2$);
else if ($bset == bset_2$) AddInclusion($bset_2, bset_1$);
else if there exists already set $\equiv bset$
if ($set \not\subset bset_1$) AddInclusion($set, bset_1$);
if ($set \not\subset bset_2$) AddInclusion($set, bset_2$);
else add $bset$ to the collection;
set $bset \subset bset_1$; set $bset \subset bset_2$;
}

The partitioning algorithm is exemplified for the simple SILAGE code in Fig. 1a. Initially, the inclusion graph of signal A contains only the nodes labeled from a to g , corresponding to the operand/definition domains (indicated as $A[\square]$) extracted from the RMSP algorithm. The first execution of the *while* loop will add to the graph the new vertices h, i, j, k, l , along with the following sequence of arcs: $(h, a), (h, b), (i, a), (i, d), (j, b), (j, e), (k, c), (k, d), (l, d), (l, c)$. Also the inclusion arcs $(e, a), (f, b)$, and (g, c) are added. The second execution of the outer loop will create the new arcs (j, h) and (l, i) , while eliminating (j, b) and (l, d) . The latter is done in the procedure *AddInclusion*, which creates new inclusion relations but deletes the transitive arcs. The final inclusion graph is shown in Fig.1b.

The partitioning into non-overlapping basic sets is derived from the inclusion graph once also the size of the sets of signals corresponding to the vertices is known. The nodes of the inclusion graph have the nice property that they represent *affine images of polytopes*. To compute their

```

#define n 6
#define W fix<16,4>

func main (in : W[n]) out : W[][]=
begin
  (j : 0 .. n-1) ::
  begin
    A[j][0] = W(0);
    (i : 0 .. n-1) ::
      A[j][i+1] = A[j][i] + W(1);
  end;
  (i : 0 .. n-1) ::
  begin
    alpha[i] = A[i][n+1];
    (j : 0 .. n-1) ::
      A[j][n+1+i] =
        if (j < i) -> A[j][n+i]
        || A[j][n+i] + 1
        fi;
  end;
end;
end;

```

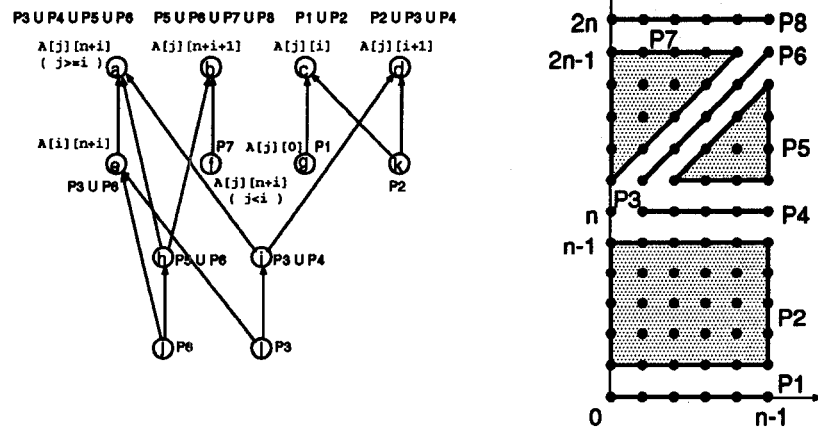


Figure 1: Illustrative example: (a) Silage code (b) the inclusion graph (c) the basic sets of signal A

size, a novel algorithm – combining Hermite normal form and Fourier-Motzkin elimination [3] – was developed.

When the affine function T in (1) is injective, counting the lattice points inside the *image* of a polytope is equivalent to counting the points of integer coordinates inside the polytope. In the sequel, a simplified version of our algorithm – handling this special case is described. The full algorithm will be presented elsewhere.

```

int CountLatticePoints(A,b)// The given polytope is  $Ax \geq b$ 
if (A.nCol==1) return Range(A,b); // handles  $ax \geq b$ 
if (FourierMotzkinElim(A,b) <= 0) return cod_error;
// cases  $Ax \geq b$  an unbounded polyhedron,
// or an empty set are detected; otherwise
// the range of  $x_0$  – the first element of  $x$  – is returned
N = 0;
for (int m = min(x0); m <= max(x0); m++)
  N += CountLatticePoints(A0, b-m*A.col(0));
// A=[A.col(0) | A0]
return N;

```

The drawback of this algorithm is its worst-case exponential time complexity. However, it has yielded very promising results so far for our practical applications (see Section 5). This is especially due to the fact that the method does not need to be applied on the constraint matrix (A in (1)) directly. Indeed, the linear constraints of a basic set are derived from the constraints on loop boundaries and control-flow conditions. As loop boundaries are frequently constant (at least the outermost loops have constant boundaries), the Fourier-Motzkin elimination is usually applied only on a *reduced* constraint matrix.

The final partitioning for our illustrative example is shown in Fig. 1c. The decomposition of the sets of signals corresponding to the nodes of the inclusion graph is indicated in Fig. 1b by the unions associated to the nodes.

4 The "quantitative" data-flow analysis

Like in any data-flow analysis, a data-flow graph is produced (Fig. 2a). However, unlike the classic case, the nodes and edges in the graph do not correspond to individual variables/signals and their dependences, but to relations between groups of signals covered by basic sets. This feature allows a successful handling of large examples.

In our approach, the goal is to obtain an accurate evaluation of the minimal memory size compatible with the

given loop hierarchy. Therefore, a traversal policy of the data-flow graph encouraging the minimization of storage locations has been employed. This favours the sequential execution of operations on signal when they can be stored in-place.

The method described so far operates on basic sets obtained from a *coarse-grain* partitioning – using the operand and definition domains directly derived from the loop organization and partial ordering provided in the initial code. If desired, the effect of the partial ordering embedded in the initial loop hierarchy can be gradually removed by means of a *finer-grain* partitioning. This can be achieved by descending in each loop nest until a predefined depth is reached, while projecting the definition/operand domains – equivalent to implicitly unrolling the loops up to that depth. The basic sets are then newly derived and partitioned corresponding to that level. Fig. 2b shows the same DFG expanded 1 loop level. There are two contrasting objectives in this case. On one hand, the number of basic sets is increasing which is an undesirable effect due to the growth of the computational effort. Moreover, also the controller complexity to be realized will grow exponentially. On the other hand, the memory size will gradually approach the absolute lower bound which is obtained if all loops are unrolled and only scalars remain. By increasing the level up to the maximum depth of the loop nests in the algorithm, the *classic data-flow analysis* would be obtained, where basic sets reduce to individual (scalar) signals. In practice, the latter case should be avoided though. Instead, for each application the granularity of basic sets can be gradually decreased, until a good trade-off is obtained. A trace of the two DFGs (Fig. 2a,b) traversal is shown in Fig. 2c.

5 Overview of main results

The implementation of the presented approach was done in C++, under the CATHEDRAL framework. It was tested on a medium-size DECstation 3100.

The Fourier-Motzkin based counting technique proved to be relatively expensive when applied directly on the constraint matrices of the basic sets. E.g., the lattice points of one test polytope containing about 1.2 billion points, were counted in 30 seconds. However, for most practical cases, it was shown in Section 3 that the constraint matrix can be reduced. As a result, the elapsed times were at least one order of magnitude smaller.

The first group of entries in Table 1 show the results

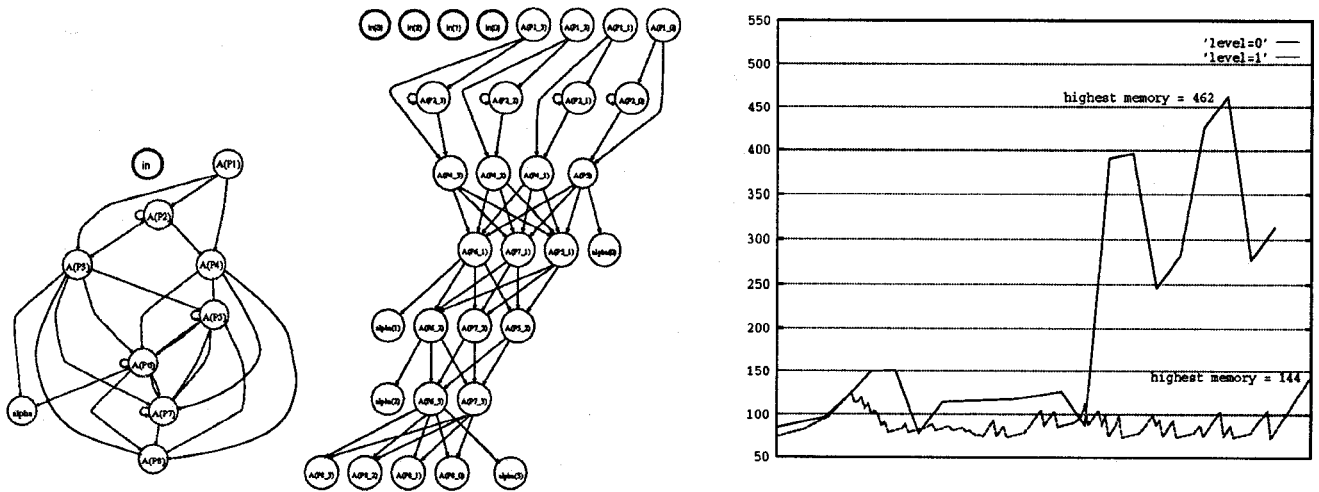


Figure 2: Data-flow analysis (a) DFG for the example in Figure 1 (b) DFG expanded 1 loop level for the example in Figure 1 ($n=4$) (c) trace of memory size during DFG traversal in the SVD updating algorithm ($n=6$)

Application	Level	Signals	BasicSets	Memory
SVD updating ($N=10$)	0	3547	44	2072
	1	3538	345	400
	2	3502	1756	400
	3	3502	3502	400
Motion detection	0	25239	11	24949
	1	25239	123	2917
	2	25239	1738	1620
	3	25239	6410	1048
Vocoder kernel(*)	0	25932	36	23584
	1	25932	8325	12361
	2	25932	25932	11269

Table 1: Results for MRSP applications
(*) autocorrelation+Schur alg.+vector quantization

obtained on a singular value decomposition (SVD) updating algorithm on a matrix with size parameter N . This is an important algebraic kernel used e.g. in beamforming and Kalman filtering. The results are provided for the different levels of loop "unrolling", up to a maximal depth. The minimum storage requirements for this algorithm representation, which is $4N^2$, is already obtained starting with the first level of granularity. This shows that total loop expansion is not a necessary condition for reaching the absolute lower bound. The method has also been evaluated on a video coding example - where a motion detection algorithm was analyzed - and for the kernel of a complex voice coding application - essential component of the mobile radio terminal. The results are indicated in the second and the third groups of entries in Table 1.

6 Conclusions

In this paper, we have addressed the important problem of background memory allocation for real-time MD signal processing systems. In this context, we have advocated the necessity of having a good evaluation of the global memory size for a given high-level *non-procedural* specification of a DSP algorithm. In the paper, we have presented a novel technique founded on data-flow analysis, which allows to address the problem of global memory

size evaluation while operation scheduling is still unknown. In most MD signal processing examples, the number of signal instances is huge, so a flattened data-flow analysis is impossible. Consequently, a new data-flow model grouping scalar signals in so-called *basic sets* is proposed. Based on this, a novel approach for memory size evaluation has been discussed. Actually, the method also incorporates a way to trade-off the memory size with the complexity of both the computational effort and the controller to be realized. These claims are substantiated with results for realistic complex applications extracted from video coding and beamforming (e.g. for mobile radio systems). The results presented in this paper constitute the foundation for our future work on solving the complete *memory allocation & port binding* optimisation problem.

References

- [1] I.Ahmad, C.Y.R.Chen, "Post-processor for data path synthesis using multiport memories," *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pp.276-279, Nov. 1991.
- [2] F.Balasa, F.Franssen, F.Catthoor, H.DeMan, "Transformation of Nested Loops with Modulo Indexing to Affine Recurrences," submitted to *Parallel Processing Letters*.
- [3] G.Dantzig, B.Eaves, "Fourier-Motzkin Elimination and Its Dual," *J. Combinatorial Theory*, Vol.14, pp.288-297, 1973.
- [4] G.Goossens, "Optimisation Techniques for Automated Synthesis of Application-Specific Signal-Processing Architectures," Doctoral dissertation, K.U.Leuven, Belgium, 1989.
- [5] F.J.Kurdahi, A.C.Parker, "REAL: a program for register allocation," *Proc. 24th ACM/IEEE Design Automation Conf.*, pp.210-215, June 1987.
- [6] P.Lippens *et al.*, "PHIDEO: a silicon compiler for high speed algorithms," *Proc. EDAC*, pp.436-441, Feb. 1991.
- [7] J.M.Mulder, N.T.Quach, M.J.Flynn, "An Area Model for On-Chip Memories and its Application," *IEEE J. Solid-state Circ.*, Vol.SC-26, pp.98-105, Feb. 1991.
- [8] M.v.Swaaij, F.Franssen, F.Catthoor, H.De Man, "High-level modelling of data and control flow for signal processing systems," in *Design Methodologies for VLSI DSP Architectures and Applications* (ed. M.Bayoumi), Kluwer, 1993.