Background Memory Area Estimation for Multidimensional Signal Processing Systems

Florin Balasa, Francky Catthoor, Member, IEEE, and Hugo De Man, Fellow, IEEE

Abstract— Memory cost is responsible for a large amount of the chip and/or board area of customized video and image processing system realizations. In this paper, we present a novel technique—found on data-flow analysis—which allows to address the problem of background memory size evaluation for a given nonprocedural algorithm specification, operating on multidimensional signals with affine indexes. Most of the target applications are characterized by a huge number of signals, so a new polyhedral data-flow model operating on groups of scalar signals is proposed. These groups are obtained by a novel analytical partitioning technique, allowing to select a desired granularity, depending on the application complexity. The method incorporates a way to tradeoff memory size with computational and controller complexity.

I. INTRODUCTION

SPEECH, image, and video processing applications involve a large amount of multidimensional signals which lead to large memory units. These result in significant area and power consumption cost in (application-specific) architectures for real-time multidimensional signal processing (RMSP), in most cases dominating the data-path contribution for complete systems [34], [20]. A 128 kb embedded SRAM takes about 115 mm² in a 1.2 µm CMOS technology, as compared to 5–15 mm² for an entire (very) complex data-path. The difference in importance would increase further if the data-paths in an architecture design would be optimized without considering the implications on memory. For these reasons, we have opted to optimize first the high-level storage organization for the multidimensional signals in our CATHEDRAL script [36]. Note that this high-level memory management stage is fully complementary to the traditional high-level synthesis step known as “register allocation/assignment” [31], [19], [15], [1]. [29] 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 [15].

Three main partly conflicting objectives can be identified during high-level memory management, solvable within different steps: 1) optimizing the memory access by allocating a number of memory units of specific types and by distributing and organizing the signals efficiently over a set of background memories and ports (referred to in the sequel as background memory allocation), 2) reducing the memory size by in-place storage of signals, and 3) minimizing the area overhead of address generation circuitry by optimizing the memory access sequences. Before these steps, we have added a global loop hierarchy optimization technique [36], based on abstract cost measures, providing constraints for the subsequent steps.

The type of control and data flow constructs, handled during memory management, mainly include irregularly nested loops and indexed signals. Our model is intended for multidimensional (delayed) signals with complex affine index expressions, and nests of loops having as boundaries affine (linear of each variable) functions with integer coefficients of the surrounding loop iterators. The functional specifications can contain condition expressions—data-dependent (i.e., dependent on produced/input signals) or not (dependent only on iterators). This covers the majority of the RMSP application domain.1 Pointers, interrupts, data-dependent loop-bounds and indexes are not incorporated in the scope of this work.

The problem of memory size estimation, tackled in this paper, is formulated as follows: “Given a RMSP algorithm described in a high-level applicative language, containing a large amount of scalars, organized mainly in multidimensional signals and assigned to a single multiport memory unit, estimate the background memory (necessary to store the multidimensional signals) area, subject to given throughput and loop organization constraints.” Different from past approaches, requiring that detailed operation scheduling has been previously performed, our formulation disregards this essential requirement, tackling the problem under more general conditions. The motivation for this is given in Section II.

The memory size estimation kernel as developed here, is useful in two different contexts. This research was done as part of solving, under the same application domain assumptions, the more general problem of background memory allocation [4]. The estimation model is targeted to (multiport) random access memories. Allocation aspects like memory hierarchy, bandwidth, latency, are beyond the scope of this paper. However, the method proposed here is also applicable in other system exploration contexts. In particular, if several descriptions of an algorithm, or several alternative algorithms for a given RMSP application, have to be traded off against one another, it is important to evaluate their memory cost in a very early system design stage. In summary, the ability to predict memory characteristics of designs without synthesizing

Manuscript received December 23, 1993; revised November 14, 1994. This work was supported in part by the ESPRIT Basic Res. Action 3281 (NANO II) project of the E.C.

The authors are with the IMEC, Kapeldreef 75, B-3001 Leuven, Belgium. IEEE Log Number 9410831.

1 However, extensions to an even more general modeling have been described by us too [11].

1063-8210/95$04.00 © 1995 IEEE
them is vital to producing high quality designs in a reasonable time.

II. BACKGROUND MEMORY SIZE ESTIMATION: CONTEXT AND STATE-OF-THE-ART

To our knowledge, almost all techniques for dealing with the allocation of storage units are scalar-oriented and employ a scheduling-directed view (see e.g., [1], [2], [19], [29], [31]) where the control steps of production/consumption for each individual signal are determined beforehand. This applies also for memory/register estimation techniques (see, e.g., [18], [13] and their references). This strategy is mainly due to the fact that applications targeted in conventional high-level synthesis contain a relatively small number of signals (at most of the order of magnitude $10^3$). The control/dataflow graphs addressed are mainly composed of potentially conditional updates of scalar signals. Therefore, as the major goal is typically the minimization of the number of registers for storing scalars, the scheduling-driven strategy is well-fitted to solve a register allocation problem, rather than a background memory allocation problem. In that case, (binary) ILP formulations [1], [2] and graph colouring [29] or clique partitioning [31] techniques have provided satisfactory results for register allocation, signal-to-register and signal-to-port assignments, under the usually implicit assumptions mentioned above.

For many of our targeted applications, where memory is the dominating resource, this general strategy—imposing a strict precedence between scheduling and storage management—can lead to costly global solutions, as the search space for memory optimization becomes heavily constrained after scheduling. Manual experiments carried out on several test vehicles [34]—as autocorrelation computation, singular value decomposition, solving Toeplitz systems—have shown that memory management decisions, taken before scheduling, can reduce significantly the background memory cost, while the freedom for data-path allocation and scheduling remains almost unaffected [6]. Furthermore, within the scheduling-directed view, many examples are untractable because of the huge size of the ILP formulations. Exceptions to this are PHIDEO [20]—where streams are used during the memory allocation but which is still after scheduling, and recently also MeSA [26]—where memory allocation cost is based both on a layout model and on the expected performance, but where the possibility of signals to share common storage locations when their life times are disjoint is neglected.

A behavioral description like that of Fig. 1 is untractable by means of a classical scalar-oriented technique. First, this class of examples usually contains large amounts of scalar signals. At the same time, scalar-oriented approaches entail a loss of the code regularity. According to our experience, this leads to an unacceptable growth of the controller size. In addition, a behavioral description language like SILAGE [17] is by definition nonprocedural: besides the natural production/consumption order imposed by the dependence relations existent in the code, there is much freedom left in the execution ordering. This can be exploited by a designer in order to meet his goals—e.g., to take profit of the parallelism “hidden” in the code. The current in-place mapping and address generation tools of the CATHEDRAL system [33] cannot handle the example of Fig. 1 if, e.g., lines (1) and (5) are interchanged, as they interpret the source code procedurally; the operation ordering taken into account is only that one from the code, where all signals must be produced before being consumed. Moreover, the allocation is done manually in CATHEDRAL-2/3.

The allocation tool MEDEA from the PHIDEO system [20] meets also problems in dealing with examples like that of Fig. 1. It is partly capable of handling nonprocedural applications, and the stream model is very well suited and yields good results for applications where the dependence vectors [5] have constant elements, as in many front-end video applications. By means of a hierarchical stream model, it is possible to handle multidimensional signals with complex affine indexes, but only in nests of loops with constant boundaries. The PHIDEO stream model is not tuned to image, speech or medium-level video processing systems which typically do not exhibit fixed periods and fixed length streams. Hence, handling loop nests with nonconstant boundaries, and nonuniform dependence relations between signals is not always possible.

Background memory size estimation has only recently gained attention in high-level synthesis. The first estimation methods are based on symbolic evaluation—a scalar-oriented technique—which consists in enumerating all indexed signals for all index combinations [24].

More recently, novel results have been obtained for the case when the algorithm specification is nonprocedural. Modifications of the loop hierarchy and the sequence of execution as specified in the source code are used to optimize the storage requirements [36]. Retaining only the data flow constraints, a new control flow is provided by placing polytopes of signals in a common space (with an ILP technique), and searching for an ordering vector in that space. For solving the memory size estimation problem in the nonprocedural case, data-flow analysis has been consistently employed by us as the main exploration strategy [3]. In a recent work, data dependence information provided by the Omega test [25] has also been

\begin{verbatim}
# define M 1024 /\* M = N /\* M = N are predefined constants */

fun main(A: [MxN]), [N:K]), [N:K]) end: M =

begin:
  caseprN1[N:K] = A[K].end:
begin:
  n = [N:K].end:
end;

Fig. 1. SILAGE code extracted from the motion detection algorithm.

\end{verbatim}
applied in memory size estimation [35]. Although the method is very appealing in terms of speed, the assumptions regarding loop hierarchy and control flow—e.g., a nest of loops is executed in the sequence as specified in the source code, and a sequence of loops is executed in the order specified in the source code—only partly remove the procedural constraints.

A precise evaluation of the memory area implies two distinct aspects:

1) assessment of the number of storage locations, which determines the total number of memory cells;
2) assessment of the number and type (read, write, or read/write) of memory ports, which heavily influences the area cost of a single cell.

Finding the minimum number of memory locations/registers when the operation ordering is that one provided by the order of statements, and the nesting of the loops in the algorithm code (or when the scheduling of the operations was previously accomplished), has already been solved. Knowing the detailed sequence of operations entails precise information concerning the life times of variables (signals). Consequently, a "left-edge" type algorithm [15], [19] is sufficient to find out in $O(n \log n)$ time, both the minimum number of storage requirements and the assignment of individual signals to memory locations. Extensions to deal with cyclic graphs have been proposed also in [15], [29]. Unfortunately, the same problem becomes significantly harder when the operation ordering is still not fixed, belonging to the NP class [14] even in the absence of conditionals. A straightforward way of estimating the number of memory locations when the operation scheduling is still not decided is by means of an accurate data-flow analysis.

In parallel compiler theory, data dependence analysis has been a topic of research for a long time. Transforming programs so as to make efficient use of massively parallel machines is a very demanding task. Determining whether a dependence exists between two array references (see [5] for a good overview) is a crucial issue for doing code transformations. Methods based on parametrized integer programming [12], and Fourier-Motzkin algorithm [9] are extensively used [25]. Data dependence has been employed to maximize the fine- or coarse-grain parallelism in loop nests, or to improve data locality [38].

As RMSP algorithms contain usually a huge amount of scalars, a data-flow analysis operating with groups of signals (rather than a flattened one operating with individual signals) is compulsory. On the other hand, doing a sufficiently accurate memory size estimation irrespective of an operation ordering, and when flattening is impractical, requires more data-flow information than the existence of dependences between array references [25] (as in code restructuring). In the SILAGE code of Fig. 1, the decision regarding the memory requirement e.g., for signals Delta cannot be taken only by determining the existence of data dependences between the 4 array references of Delta in lines (2)-(4). It can be proven, employing the number of dependences between different parts of the arrays, that $(M - m + 1)(N - n + 1)$ is the best memory size upper-bound. Basing the decisions only on the existence of data dependences can lead to an important over-estimation/allocation of memory. These aspects led to the development of an accurate analysis of the dependence structure, operating on a polyhedral dependence graph model.

This paper presents a nonscalar method for estimating the area of the background memory, before operation scheduling, for RMSP algorithms with nonprocedural specifications. This technique can then be used to guide the distributed memory allocation [4] or early system-level exploration tasks. In order to address applications with large amounts of scalars, an analytical partitioning of the indexed signals will be presented in Section III. A polyhedral dependence graph model—built around the concept of basic set of signals—will be presented in Section IV. The assessment of the number of storage locations for algorithms described in a nonprocedural language is presented in Section V. Results obtained so far are substantiated in Section VI, followed by the conclusions and our future directions of research in Section VII. In the sequel, matrices and vectors are denoted with bold characters, matrices being in capitals.

III. ANALYTICAL PARTITIONING
OF MULTIDIMENSIONAL SIGNALS

The set of appearances of an indexed signal in the RMSP algorithm is characterized by the collection of definition (left-hand side) and operand (right-hand side occurrences) domains for that signal [36]. Each domain has an index space which is a linearly bounded lattice (LBL) [30]—the image of an affine mapping over a set of linear inequalities representing a polytope:

$$\{ x = T \cdot i + u | A \cdot i \geq b \}$$

where $x \in Z^n$ is the coordinate vector of an $m$-dimensional signal, and $i \in Z^m$ is the vector of loop iterators. The affine function is characterized by $T \in Z^n \times n$ and $u \in Z^n$, while the integral polytope—defining the set of iterator vectors—is characterized by $A \in Z^{2n \times n}$ and $b \in Z^{2n}$. For instance, the index space of the operand domain $Delta[i][j][2 * m + 1] * (2 * n + 1)$ in line (4) from the SILAGE code in Fig. 1 is represented by the following:

$$\begin{bmatrix}
  x \\
  y \\
  z
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & i \\
  0 & 1 & j \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  0 \\
  0 \\
  (2m + 1)(2n + 1)
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & i \\
  -1 & 0 & j \\
  0 & 1 & 1 \\
  0 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
  m \\
  n \\
  -M \\
  -N
\end{bmatrix}.$$

In the sense that there is an operation ordering requiring $(M - m + 1)(N - n + 1)$ locations, no ordering requiring more. Of course, there are orderings needing less memory.

In general, the index space may be a collection of linearly bounded lattices. For example a conditional instruction $if (i \neq j)$ determines two LBL’s for the domains of signals within the scope of the condition—one corresponding to $i \geq j + 1$, and another corresponding to $i < j + 1$. Without any decrease in generality, we can assume in the sequel that each index space is represented by a single linearly bounded lattice.
Once the collections of definition/operand domains are extracted from the signal processing algorithm, a partitioning process into nonoverlapping "pieces"—called basic sets in the sequel—is performed for each collection. The motivation is the following: in order to derive dependence relations between groups of signals, it is necessary to know which part of an operand is produced by which definition domain, and which part of a definition is consumed by which operand domain. This information is needed for memory estimation (see Section V).

The signals common to a given set of domains and only to them will constitute a basic set. For example, a collection of 2 domains results in at most 3 basic sets; 3 overlapping domains result in maximally 7 sets. More general, if the number of definition/operand domains of a multidimensional signal is denoted by \( s \), the number of basic sets is upper-bounded by \( C_s^2 + C_s^3 + \ldots + C_s^s = 2^s - 1 \). However, this evaluation proved to be extremely pessimistic in practical cases. In the illustrative example in Fig. 2(a), for instance, the number of domains for signal \( A \) is \( s = 9 \), and the resulting number of basic sets is only 10 (see Section III-B).

A. The Partitioning

The aim of the partitioning process is to determine what parts of an operand domain are not needed any more after the computation of a definition domain, assuming a given loop hierarchy and a certain data-flow. In other words, which are the scalar signals consumed for the last time, and what is their number, when the signals within a definition domain are produced. The last question is related directly to the evaluation of the storage requirements, as it allows to compute exactly how many memory locations are needed and how many can be freed when a certain group of signals is produced, assuming a certain data-flow. Furthermore, the subsequent partitioning of the indexed signals, together with dependence information, can offer solutions for partially rearranging the loop organization provided in the initial code, in order to obtain a decrease of the storage requirements (see Section V).
The partitioning process—described in the sequel—yields a separate inclusion graph for each of the collections of linearly bounded lattices. The direct inclusion relation is denoted by "⊂," while its transitive closure (the existence of a path in the graph) is denoted by "≺." An inclusion graph is constructed as shown in Algorithm 1 on the bottom of the page.

The procedure AddInclusion creates new inclusion relations (arcs) between groups of signals, but deletes the resulting transitive arcs: keeping a strict hierarchy for the LBL's is essential for the partitioning phase. The intersection "∩" of two linearly bounded lattices is described in Section III-C. An example will be provided in Section III-B.

Afterwards, the basic sets of a signal are derived from the inclusion graph with a simple bottom-up technique as shown in Algorithm 2 on the bottom of the page.

If a node has no components, a new basic set—equal to the corresponding linearly bounded lattice—is introduced.

Two linearly bounded lattices of the same indexed signal are equivalent if they represent the same set of indices. For example, \( \{x = i + j | 0 \leq i \leq 2, 0 \leq j \leq 2\} \) and \( \{x = i | 0 \leq i \leq 4\} \) are equivalent. Testing LBL's equivalence can be done employing LBL intersection and size determination (see Section III-C and -D).

otherwise, if all its components have already been partitioned, the union of the component partitions and (potentially) a new basic set will constitute the current node partitioning. In the latter case, the new basic set appears only if there is a difference between the size of the node and the total size of its components.

The computation of LBL sizes is described in Section III-D. The efficiency of this operation and of the intersection procedure are crucial for the practical time complexity of the whole algorithm. The partitioning process is exemplified in Section III-B.

The importance of this analytical partitioning into groups of signals—called basic sets—is manifold.

1) It allows a quick verification (without descending at the scalar level) whether the program complies with the single assignment conjecture: it is sufficient to check that each basic set belongs to one single definition domain.

2) It allows a quick detection and elimination of obsolete signals in the program.

3) It allows the estimation of the memory size by means of a data-flow analysis (see Section IV) at basic set level (rather than scalar level), as shown in Section V.

---

**Algorithm 1:**

```java
void ConstructInclusionGraph(collection of LBL's—the index spaces of signal domains) {
    initialize inclusion graph with the def/opd index domains as nodes;
    while new nodes/arcs can be appended to the graph
        for each pair of LBL's (Lbb1, Lbb2) in the collection
            such that (Lbb1 ≠ Lbb2) && (Lbb2 ≠ Lbb1) {
                compute Lbl = Lbb1 ∩ Lbb2;
                if (Lbl ≠ ∅)
                    if (Lbl==Lbb1) AddInclusion(Lbb1, Lbb2);
                    else if (Lbl==Lbb2) AddInclusion(Lbb2, Lbb1);
                    else if there exists already lbl ≡ Lbl 6
                        if (lbl ≠ Lbb1) AddInclusion(lbl, Lbb1);
                        if (lbl ≠ Lbb2) AddInclusion(lbl, Lbb2);
                    else add Lbl to the collection;
                    set Lbb1 ⊂ Lbb1; Lbb2 ⊂ Lbb2;
            }
}
```

**Algorithm 2:**

```java
void CreateBasicSets(inclusion graph) {
    for every LBL (node in the inclusion graph) having no arc incident to it
        create a new basic set P, equal to that LBL;
    while there are still non-partitioned LBL's {
        select a nonpartitioned LBL (node in the inclusion graph) such that
        all arcs incident to it emerge from partitioned LBL's (LBLi = \( U_j P_{ij} \));
        if (size(LBL) > size(\( U_i LBLi \)))
            create a new basic set P equal to LBL \( \setminus U_i LBL_i \);
        partition the LBL into \( U_i U_j P_{ij} \);
        else // size(LBL) = size(\( U_i LBL_i \))
            partition the LBL into \( U_i U_j P_{ij} \);
    }
```
B. An Illustrative Example

The partitioning algorithm is exemplified for the simple SILAGE code in Fig. 2(a). Initially, the inclusion graph of signal A contains only the nodes labelled from a to i, corresponding to the operand/definition domains (indicated as A[|]) extracted from the source code. The first execution of the while loop in Algorithm 1 adds to the graph the new vertices j, · · · , r, along with the sequence of arcs:

\((j, b), (j, c), (k, b), (k, d), (l, b), (l, h), (m, d), (m, g), (n, e), (n, f), (o, c), (o, i), (p, f), (p, g), (q, g), (q, h), (r, g), (r, i)\).

Also the inclusion arcs \((d, h)\) and \((a, c)\) are added. The second execution of the while loop creates the new arcs \((m, q)\) and \((k, l)\), eliminating \((m, g)\) and \((k, b)\) at the same time. The latter is done in the procedure AddInclusion, which creates new inclusion relations, deleting the transitive arcs. For example, a node \(l\) represents the LBL

\[
\begin{bmatrix}
  x \\
  y
\end{bmatrix} = \begin{bmatrix}
  0 & 1 \\
  0 & 0
\end{bmatrix} \begin{bmatrix}
  i \\
  j
\end{bmatrix} + \begin{bmatrix}
  0 \\
  0
\end{bmatrix} \begin{bmatrix}
  0 & 1 \\
  -1 & 0
\end{bmatrix} \begin{bmatrix}
  i \\
  j
\end{bmatrix} \geq \begin{bmatrix}
  0 \\
  -n + 1
\end{bmatrix}
\]

resulting from the intersection of the linearly bounded lattices corresponding to nodes \(h\) and \(b\) shown at the bottom of the page.

The final inclusion graph is shown in Fig. 2(b).

The basic sets are derived bottom-up, after the computation of all node sizes in the inclusion graph, with the technique described in Algorithm 2 (Section III-A). The BRL's in the inclusion graph are partitioned in the order: a, j, k, m, p, n, o, r, c, d, e, f, i, l, q, b, g, h. The final partitioning for our illustrative example is shown in Fig. 2(c). Each domain has an index space which consists in one (e.g., \(A[j][0] \rightarrow A0\)) or several basic sets (e.g., \(A[j][i] \rightarrow A0 \cup A1\)). For instance, the scalars produced by \(A[j][i+1]\) and consumed by \(A[i][n + i]\) are represented by their intersection which corresponds to the basic set \(A2\).

Remark Basic sets cannot be always represented as linearly bounded lattices (as it is the case in this illustrative example). In general, they can be decomposed, or they can be expressed as differences of affine mappings of polytopes. For example, signal A from the SILAGE code in Fig. 1 has two basic sets; one of them—\(A1\)—cannot be represented as a linearly bounded lattice (see Fig. 4(b)).

C. Intersection of Linearly Bounded Lattices

Let \(\{x = T_1 i_1 + u_1 | A_1 i_1 \geq b_1\}, \{x = T_2 i_2 + u_2 | A_2 i_2 \geq b_2\}\) be two LBL's derived from the same indexed signal where \(T_1\) and \(T_2\) have obviously the same number of rows—the signal dimension. Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system \(T_1 i_1 - T_2 i_2 = u_2 - u_1\) having the elements of \(i_1\) and \(i_2\) as unknowns. If the system has no solution, the intersection is empty. Otherwise, let

\[
\begin{bmatrix}
  i_1 \\
  i_2
\end{bmatrix} = \begin{bmatrix}
  V_1 \\
  V_2
\end{bmatrix} \begin{bmatrix}
  i \\
  v_2
\end{bmatrix}
\]

be the solution of the Diophantine system. If the set of coalesced constraints

\[
\begin{align*}
A_1 V_1 \cdot i & \geq b_1 - A_1 v_1 \\
A_2 V_2 \cdot i & \geq b_2 - A_2 v_2
\end{align*}
\]

has at least one integer solution, then the intersection is a new linearly bounded lattice defined by \(\{x = T \cdot i + u | A \cdot i \geq b\}\) where

\[
T = T_1 V_1, \quad u = T_1 v_1 + u_1 \\
A = A_1 V_1, \quad b = b_1 - A_1 v_1.
\]

Solving a linear Diophantine system was proven to be of polynomial complexity [27]. All the known methods are based on bringing the system matrix to the Hermite Normal Form [27]. They only differ from each other basically by the modality in which this representation is obtained. Numerous researchers proposed algorithms for computing Hermite and Smith Normal Forms, as well as the related problem of solving systems of linear Diophantine equations. After 1977, more recent results have been obtained, overcoming the main shortcoming of the classical methods, namely the "intermediate expression swell." A pathological example substantiating this effect is given in [16]. Therefore, several algorithms with provable polynomial worst-case complexity have been proposed (e.g., [16], [27]). The size of Diophantine systems in most of our practical cases does not justify the overhead (in programming and practical computation effort) implied by the use of one of those more sophisticated algorithms with provable polynomial worst-case complexity. Therefore, we have searched for a simpler technique which works well for smaller problem sizes.

In our current implementation, each unimodular transformation reduces the coefficients of an equation (except the smallest one) to at most 1/2 of the smallest coefficient, therefore employing usually a shorter sequence of unimodular transformations than [22]—where the reduction factor is 1, or [25]—where the reduction factor is 2/3. The basic idea is the
following. Consider the Diophantine equation $a_{11}x_1 + \cdots + a_{1n}x_n = a_1 \in \mathbb{Z}$, where $a_1 \in \mathbb{Z} - \{0\}$, and suppose, without loss of generality, that $a_{11}$ is one of the coefficients such that $a_{11} = \min \{|a_{11}|, \ldots, |a_{1n}|\}$. In the unimodular substitution $x_1 = X_1 + c_2x_2 + \cdots + c_nx_n$, the coefficients $c_k$ can be chosen to be integers satisfying $|a_{11}c_k + a_1k| \leq \frac{1}{2} |a_{11}|$, $k = 2, \ldots, n$:

$$
c_k = \begin{cases} 
-\text{sgn}(a_{11})|a_{1k}|/|a_{11}|, & \text{if } 0 \leq a_{1k} - |a_{11}||a_{1k}/|a_{11}|| \leq |a_{11}|/2 \\
-\text{sgn}(a_{11})(|a_{1k}/|a_{11}|| + 1), & \text{if } |a_{11}|/2 < a_{1k} - |a_{11}||a_{1k}/|a_{11}|| < |a_{11}|.
\end{cases}
$$

An undesirable side-effect of intersection is the rapid size increase of the polytope description for the resulting linearly bounded lattice, due to the coalescing of the two constraint sets. Therefore, minimizing the set of constraints proved to be a necessity in order to restrict the computational effort of, e.g., counting the lattice points of the resulting LBL's (see Section III-D). The technique employed for minimizing the set of linear constraints is based on Chernikova's algorithm [8]—a nonpivoting method for finding all vertices of convex polytopes. The implementation in use stems from the polyhedral library developed at IRISA [37].

**D. Computation of the Affine Image Size of a Polytope**

When the linear function $t: \mathbb{Z}^n \rightarrow \mathbb{Z}^n$, defined by $t(\bar{a}) = T\bar{a}$ is injective, the number of lattice points$^7$ of the image of a polytope is equal to the number of points of integer coordinates inside the polytope. The latter problem was tackled long ago, but only recently it was proven [10] the existence of a polynomial-time solution up to the 4-D case.

As a more general solution—able to handle signals of any dimension—was needed, a novel technique based on the Fourier-Motzkin elimination [9] has been developed. The routine for counting the lattice points inside a given polytope is described below. In the sequel, the columns of a matrix are denoted by subscripted vectors, e.g., $A = [a_0, a_1, \ldots]$; the number of columns is denoted by $A.nCol$ (see Algorithm 3 at the bottom of the page).

The main idea of the algorithm (shown at the bottom of the page) is the following: the number of lattice points in the given polytope, having as first coordinate $z_0$, is equal to the number of lattice points in the polytope $a_1z_1 + a_2z_2 + \cdots \geq b - a_0z_0$ (which has one dimension less). The required result is obtained by accumulating this number over the entire range of $z_0$ (determined by Fourier–Motzkin elimination). The worst case time complexity of the algorithm is exponential. The influence of this negative aspect is attenuated by minimizing first the number of inequalities with Chernikova’s algorithm.

In addition, 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, the Fourier–Motzkin elimination is usually applied only on a reduced constraint matrix. Moreover, the initial number of columns of matrix $A$ is the number of the surrounding loops, which is usually small.

Two questions arise: how can it be decided whether function $t$ is injective or not, and what is to be done if the injectivity condition is not fulfilled. In this context, it has to be emphasized that generating all lattice points in the polytope and collecting their images in a set is very inefficient for “large” polytopes.

For any matrix $T \in \mathbb{Z}^{m \times n}$ there exists a unimodular matrix $S \in \mathbb{Z}^{n \times n}$ such that (less a row permutation)

$$
\begin{bmatrix}
H_{11} & 0 \\
H_{21} & 0
\end{bmatrix},
$$

where $H_{11}$ is nonsingular lower triangular. Such a unimodular matrix can be computed with the technique, briefly described in Section III-C, for solving linear Diophantine systems.$^8$

Denoting $\text{rank} H_{11} = r$, and $S^{-1}i = j$, the following situations may be encountered:

1) $r = n$. The affine mapping $x = Ti + u = [H_{11} \quad j + u] \quad [H_{21}]$ is injective, as $x_1 = x_2$ implies $j_1 = j_2$ (as $H_{11}$ is lower triangular), hence $i_1 = i_2$.

2) $r < n$. Then $x_1 = x_2$ implies that only the first $r$ components of $j_1$ and $j_2$ are resp. equal. This means that all vectors $j$ satisfying $AS \cdot j \geq b$, and having the same prefix $j_1 \cdots j_r$, contribute to the image set with one single distinct value because they are mapped to the

$^7$For example, with integer coordinates.

$^8$In fact, any sequence of unimodular transformations bringing matrix $T$ to the Hermite Normal Form may be employed (e.g., [27], [51]).

---

**Algorithm 3**:

```c
int CountLatticePoints(A, b) { // The given polytope is Az ≥ b
if (A.nCol==1) return Range(A, b); // handles the trivial case az ≥ b
if (FourierMotzkinElim(A, b) ≤ 0) return code_error;
// special cases when Az ≥ b is an unbounded polyhedron, or an empty set are detected;
// otherwise, Fourier_MotzkinElim returns the range of z0—the first element of z
N = 0;
for (int z0 = min(z0); z0 ≤ max(z0); z0++)
    N += CountLatticePoints([a1, a2, ..., b - a0z0];
return N;
}
```
same point. Consequently, the affine image size of the polytope is the number of all valid prefixes \(j_1 \cdots j_r\) for which there exist vectors \(j\) having that prefix. If for each of the distinct prefixes \(j_1 \cdots j_r\), there is at most one single vector \(j\) in the polytope \(AS \cdot j \geq b\) having that prefix, the affine mapping is injective. Therefore, the complete algorithm for counting the image size of the polytope \(AZ \geq b\) becomes Algorithm 4 shown on the bottom of the page.

**Example 1:**

for \(i = 0\) to 255
for \(j = 0\) to 255
\(\cdots M[2i + 3j + 1][5i + j + 2][4i + 6j + 3] \cdots \)

\[\{x = Ti + u[Ai \geq b] = \begin{bmatrix} 2 & 3 \\ 5 & 6 \\ 4 \end{bmatrix} \begin{bmatrix} i \\ j \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ 0 \end{bmatrix} \geq \begin{bmatrix} 255 \\ 255 \end{bmatrix} \} \]

As there is a unimodular matrix

\[S = \begin{bmatrix} 1 & 3 \\ -1 & 2 \end{bmatrix} \]

such that

\[TS = \begin{bmatrix} H_{11} \\ \cdots \\ H_{21} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -4 & 13 \\ 2 & 0 \end{bmatrix}, \]

it results that \(\text{rank} H_{11} = 2 = T_{n Col}\). Hence, the domain size is equal to \(\text{size}(Ai \geq b) = 256 \times 256 = 65536\). The size of the polytope can be computed either with the routine \(\text{CountLatticePoints}(A, b)\) or, more efficiently in this case, by taking into account that both iterators have constant bounds with range 256.

**Example 2:**

for \(i = 0\) to 7
for \(j = 0\) to 7
for \(k = 0\) to 7
\(\cdots M[i + k][j + k] \cdots \)

With the unimodular matrix

\[S = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}, \quad TS = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.\]

As \(r < n(2 < 3)\), the prefixes of all vectors \(j\) satisfying \(AS \cdot j \geq b\), that is \(\{7 \geq j_1 - j_2 \geq 0, 7 \geq j_2 - j_3 \geq 0, 7 \geq j_3 \geq 0\}\), have to be checked for validity. These prefixes \((j_1, j_2)\) result to have the ranges \(j_1 \in [0, 14] \text{ and } j_2 \in [\max\{0, j_1 - 7\}, \min\{14, j_1 + 7\}]\). All 169 of them prove to be valid, hence the domain size is 169. The affine mapping is not injective, as it can be easily

---

**Algorithm 4:**

```c
int CountPoints(T, A, b) {
    compute unimodular matrix S such that T \cdot S = \begin{bmatrix} H_{11} & 0 \\ H_{21} & 0 \end{bmatrix}, \text{ with } H_{11} \text{ lower triangular;}
    let r = rank H_{11};
    if (r = T_{n Col}) return CountLatticePoints(A, b); // case 1: mapping t is injective
    else return CountPrefixes(AS, b, r); // case 2
}

int CountPrefixes(A, b, r) { // Generates potential prefixes of length r for the vectors
    // in the polytope A â¥ b. Only valid prefixes contribute to the image size
    if (r = 0) { // If the whole prefix has been generated, it is checked whether
        if (LatticePoints(A, b)) return 1; else return 0; // the prefix is valid or not
    } else if (FourierMotzkinElim(A, b)) // Az â¥ b is assumed to be a nonempty bounded polyhedron
        // returns the range of z_0—the first element of z
    N = 0;
    // for every possible first component of the prefix (of length r)
    // the rest of the prefix (of length r - 1) is generated
    for (int z_0 = min(z_0); z_0 <= max(z_0); z_0++)
        N += CountPrefixes([a_1, a_2, \cdots], b - a_0 z_0, r - 1);
    return N;
}

bool LatticePoints(A, b) { // Returns true (â‰ 0) if the polyhedron A z â¥ b is nonempty
    if (A_{n Col} == 1) return Range(A, b); // handles az â¥ b
    if (FourierMotzkinElim(A, b) == 0) return code.error;
    for (int z_0 = min(z_0); z_0 <= max(z_0); z_0++)
        if (LatticePoints([a_1, a_2, \cdots], b - a_0 z_0)) return 1;
    return 0;
}
```
seen noticing that the triplets \((i, j, k)\) equal to \((0, 1, 1)\) and \((1, 2, 0)\) are mapped to the same point.

IV. THE POLYHEDRAL DATA-FLOW ANALYSIS

After the partitioning process described in Section III, a data-flow graph (DFG) with exact dependence relations is produced (Fig. 3(a)). However, unlike the classic case of data-flow analysis, the nodes in the graph do not correspond to individual variables/signals, but to groups of signals (covered by the basic sets derived in Section III), and the arcs correspond to the dependence relations between these groups. The nodes in the data-flow graph are weighted with the size of their corresponding basic sets, and the arcs between nodes are weighted with the exact number of dependences between the basic sets corresponding to the nodes. The outcome of such a strategy is the successful memory estimation for RMSP algorithms with high amount of scalar signals (see Section VI).

Section IV-A presents the computation of dependences between the basic sets of signals. Section IV-B explains the modifications of a data-flow graph in order to deal with delayed signals, while Section IV-C deals with the data-flow analysis at different granularity levels.

A. Computation of Dependence Relations

As the nodes in the DFG and their weights—the basic sets of signals and their sizes—are known from Section III, the construction of the graph is completed by determining the arcs between the nodes—the dependence relations between the basic sets—and their weights—the number of dependences.

Suppose, without decrease in generality, that two basic sets can be represented as LBL’s:

\[
S_1 = \{x = T_1\lambda + u_1 | A_1\lambda \geq b_1\},
\]

\[
S_2 = \{x = T_2\mu + u_2 | A_2\mu \geq b_2\}
\]
and the two basic sets belong respectively to the index spaces of a definition domain and of an operand domain within the same instruction scope:

\[ D_1 = \{ x = C_1 I_1 + d_1 | A I_1 \geq b \}, \]

\[ D_2 = \{ x = C_2 I_2 + d_2 | A I_2 \geq b \}. \]

Solving the linear diophantine system in \((I_1, \lambda)\) as variables:

\[ C_1 I_1 + d_1 = T_1 \lambda + u_1 \]

and substituting the solution\(^9\) in the sets of constraints \(A I_1 \lambda \geq b_1\) and \(A I_1 \geq b\), the expression of the iterator vector corresponding to the basic set \(S_1\) is obtained

\[ \{ I_1 = T_1 \alpha + u_1 | A I_1 \geq b_1 \}. \]  

(2)

The expression of the iterator vector corresponding to the basic set \(S_2\) is obtained similarly:

\[ \{ I_2 = T_2 \beta + u_2 | A I_2 \geq b_2 \}. \]  

(3)

There is a dependence relation between \(S_1\) and \(S_2\) if there is at least one iterator vector corresponding to both of them. The number of iterator vectors yields, in this case, the number of dependences. The problem is solved by intersecting the linearly bounded lattices (2) and (3). If the intersection is empty, there is no dependence relation between \(S_1\) and \(S_2\). Otherwise, the size of the intersection (see Section III-D) represents the number of dependences.

**Example:** In the SILAGE code of Fig. 2(a), basic set \(A8\) belongs to the index space \(D8\) of the definition domain \(A[j][n + i + 1][j \geq i] \) (node \(g\) in Fig. 2(b)), and basic set \(A9\) belongs to the index space \(D9\) of the operand domain \(A[j][n + i][j \geq i] \) (node \(h\) in Fig. 2(b)). Employing a nonmatrix notation, the linearly bounded lattices of the basic sets and of the index spaces are [see Fig. 2(c)]

\[ A8 = \{ x = \lambda_1, y = \lambda_2 + n + 1 \leq \lambda_1, \lambda_2 \geq 1, \lambda_1 - \lambda_2 \geq 1 \}, \]

\[ A9 = \{ x = \mu + n | n - 1 \geq \mu \geq 1 \}, \]

\[ D8 = \{ x = j, y = i + n + 1 | n - 1 \geq j \geq 0, n - 1 \geq i \geq 0 \}, \]

\[ D9 = \{ x = j, y = i + n - 1 \geq j \geq 0, n - 1 \geq i \geq 0 \}. \]

The set of iterators corresponding to \(A8\) in the index space \(D8\) is described by \(\{i_1 = \alpha, j_1 = \alpha'n - 1 \geq \alpha, \alpha' \geq 0, \alpha'' - \alpha' \geq 2\}\). The set of iterators corresponding to \(A9\) in the index space \(D9\) is \(\{i_2 = 0, j_2 = \beta | n - 1 \geq \beta \geq 1\}\).

The intersection of the two LBU's is represented as \(\{i = 0, j = \gamma | n - 1 \geq \gamma \geq 2\}\). Hence, the number of dependences between \(A9\) and \(A8\) is \(n - 2\), i.e., the size of the intersection.

The data-flow graph of the example in Fig. 2(a) is shown in Fig. 3(a). The nodes are labelled with the signal name, basic set number, and size; the arcs are labelled with the number of dependences. \(OUT\) is a dummy node, necessary for handling delayed signals.

**B. Handling the Delayed Signals**

RMS algorithms describe the processing of streams of data samples. The source code of these algorithms can be imagined as surrounded by an imaginary loop having \(t\) as iterator. Consequently, each signal in the algorithm has an implicit extra dimension corresponding to the \(t\) time axis. RMS algorithms contain usually delayed signals, i.e., signals produced or inputs in previous data-sample processings, which are consumed during the current sample processing. The delay operator \(\oplus t\) refers relatively to the past processings. The delayed signals must be kept "alive" during several time iterations, i.e., they must be stored in the background memory during several data-sample processings.

In order to simulate the effect of the delayed signals in terms of memory requirements, a simple but effective method for handling delays is introduced in the sequel.

First, the delayed operand domains take part in the partitioning process (Section III-A) as any other signal domain. Afterwards, the construction of the data-flow graph needs the following preprocessing step:

**create a dummy node **\(OUT\)**;

**for each basic set \(b\)**

let \(d_b\) be its highest delay value (when belonging to an operand domain);

create \(d_b\) copies\(^9\) of the basic set node, each one labeled from 1 to \(d_b\);

for every copy labeled 1, \(\cdots\), \(d_b - 1\) create a

dependence from the copy node to \(OUT\);

for every basic set \(b\) belonging to an output signal, or

having a max delay \(d_b > 0\)

create a dependence arc from its corresponding node to \(OUT\);


This modification of the data-flow graph allows to take into account the effect of delays, "translating" the basic sets from previous data-sample processings into the current one. When the delay values are constant, the extension of all domains with one extra dimension—corresponding to the implicit time loop—is avoided, hence reducing the computational effort.\(^11\)

Fig. 4(a) shows the data-flow graph for the SILAGE code in Fig. 1. The basic sets corresponding to delayed signals are labelled with \(\oplus delay_value\).

**C. Data-Flow Analysis at Different Granularity Levels**

The method described so far operates on groups of signals obtained from a coarse-grain partitioning—using the index spaces of operand and definition domains directly derived from the loop organization provided in the initial code. Due

\(^9\)In practice, all copies of a basic set having a unique dependence relation (only towards the node \(OUT\)) are treated as a single item, so handling signals with high delay values does not cause computational problems.

\(^11\)Although our model allows the handling of signals with nonconstant delay values (affine functions of loop iterators), the necessity of an explicit time dimension cannot be avoided in this case.
results can be obtained, for instance, by unrolling gradually the loops. It must be also remarked that starting with a certain granularity level (dependent on the application) the DFG becomes a directed acyclic graph. Fig. 3(b) shows the same data-flow graph as in Fig. 3(a) but expanded one loop level.

There are two contrasting objectives when selecting the granularity level. On one hand, the number of basic sets is rapidly increasing, which is an undesirable effect due to the growth of the computational effort. Moreover, also the complexity of the controller in the architecture to be realized will grow exponentially. On the other hand, the memory size will gradually get closer to an absolute lower bound. In practice, descending to the scalar level ought to be avoided. Instead, for each application, the granularity should be gradually increased until a good tradeoff is obtained.

V. EVALUATION OF MEMORY SIZE

After partitioning the signals from the given RMSP algorithm, and after accumulating all the possible dependence information at the level of basic sets, the subsequent step is to obtain an accurate evaluation of the minimal memory size (locations) compatible with the resulting data-flow graph.

Even the simpler problem of finding the minimum number of memory locations necessary to compute a directed acyclic graph has been proven to be an NP-complete problem [28]. Structurally, the DFG's determined as in Section IV can be more complex, e.g., they may contain cycles, as 2 groups of signals may contain 2 subsets with opposite dependencies to the other group. Consequently, the assessment of the minimal memory size necessary to compute a data-flow graph is achieved basically by means of a heuristic traversal. It must be emphasized that the goal of this approach is to introduce only a partial operation ordering—necessary to reduce the storage requirements—while a proper scheduling implies a total ordering—which is unnecessary for our problem. The DFG traversal provides a data-flow which is equivalent to a certain reorganization of the code, without affecting the loop hierarchy. The procedural execution of this functionally equivalent code entails a low (eventually minimum) number of storage locations for computing the respective data-flow graph.

Applying the same approach to the DFG for a higher granularity level (see Section IV-C), the resulting data-flow corresponds to a certain code reorganization when the nests of loops are (implicitly) unrolled until a depth equal to the granularity level. As the granularity is increasing, the storage requirement to compute the corresponding graph is decreasing, but the number of supplementary scheduling constraints are also increasing.

Section V-A presents the basic features of the DFG traversal. Section V-B introduces the methodology of in-place mapping, incorporated in the traversal approach. The main ideas in this section will be highlighted in Section V-C, while processing the example in Fig. 1.

\[^{12}\text{In such a case, node clustering is required.}\]
A. Data-Flow Graph Traversal

The DFG traversal attempts to find the best possible ordering in which the basic sets of signals should be produced such that the memory size is kept as low as possible. It is assumed that a basic set can be produced only after the production of all basic sets having dependencies to it.

While producing a basic set, an increase of memory size occurs [see Fig. 5(a)]. This increase is followed by an eventual decrease, representing the total size of basic sets consumed for the last time (having no other dependence relations towards basic sets still not produced). The production of a basic set may not affect the current value of the maximum memory size—registered after starting the traversal [Fig. 5(a)], or may lead to exceeding this value (MaxMemorySize) which has to be therefore updated [Fig. 5(b)]. The amount of the memory is yielded by the worst-case in-place mapping (see Section V-B).

Taking into account the remarks and assumptions mentioned above, the data-flow graph traversal works according to the following general scheme shown at the bottom of the page.

This is a relatively greedy approach, but it is based on a thorough global analysis for deriving the cost factors (see also Section V-B) which decide the order of the basic set production. The results in Section VI will demonstrate the effectiveness of the traversal scheme.

B. In-Place Mapping

The problem of in-place mapping [34] refers to the possibility that signals in a RMSP algorithm share the same memory location during algorithm execution. This problem may be approached from two viewpoints:

1) high-level in-place mapping—referring to the memory sharing due to the data-flow, hence independent of any operation scheduling;
2) low-level in-place mapping—which refers to the same issue but dependent on the scheduling.

The latter problem can be easily tackled with a left-edge algorithm (e.g., [15], [19]), but the former is more difficult as it requires the computation of tight upper-bounds on the (usually huge) set of valid operation orderings.

Example: The SILAGE code in Fig. 6(a) yields the basic sets in Fig. 6(b) and the data-flow graph in Fig. 6(c). Assuming that operations are executed sequentially, and that signals A are not consumed elsewhere in the code, the problem is to find an upper-bound of the minimum number of memory locations.

A thorough analysis shows that 110 locations are sufficient, for any sequential computation of signals B[i][j]. There are orderings yielding lower memory sizes but our model re-

\[
\begin{align*}
(1: 0 .. 9):& \quad \text{begin} \\
&\quad A(2*i+1) = W(0); \quad A(2*i+1) = W(1); \\
&\quad \text{end}; \\
(1: 0 .. 9):& \quad \text{(a)} \\
&\quad B(1) = A(1+i) + A(2*i+1); \\
&\quad \text{(b)} \\
&\quad A0 = \{ x=21 \mid 9>y>1, x>0 \} \\
&\quad A1 = \{ x=21+i \mid 8>x>1, x>0 \} \\
&\quad A2 = \{ x=19 \} \\
&\quad B0 = \{ x=1, y=3 \mid 9>y>1, x>3, x>0 \} \\
&\quad \text{(c)} \\
&\quad \text{do} \quad \text{for all basic sets still not produced, but having all the ancestors produced} \\
&\quad \text{compute the cost function} \quad \text{cost} = \lambda_1 \cdot \text{peak} + \lambda_2 \cdot (\text{MemoryIncrease} - \text{MemoryDecrease}) \quad \text{where} \quad \text{peak} = \max \{0, \text{MemorySize} + \text{MemoryIncrease} - \text{MaxMemorySize}\} \\
&\quad \text{select the basic set having the minimum cost, and declared it produced} \\
&\quad \text{update the current values of MemorySize and MaxMemorySize} \\
&\quad \text{until all basic sets are produced}
\end{align*}
\]
quires upperbounds, as the script assumes that scheduling has not been decided yet (see Section II).

The formal methodology for the computation of exact memory upper-bounds—based on the dependence information embedded in the data-flow graph—will be described elsewhere. This approach has been incorporated in our memory estimation tool, in order to determine the term \textit{MemoryIncrease} from the DFG traversal scheme (see Section V-A).

C. Example of Memory Size Evaluation

The memory size evaluation methodology is exemplified for the SILAGE code in Fig. 1.

The number of memory locations required for the algorithm execution is approximated with the number of locations necessary to compute its data-flow graph for a chosen granularity level. For instance, choosing the level equal to 0, we have to compute the number of locations required for the computation of the DFG in Fig. 4.

According to the traversal scheme (Section V-A), the basic sets are produced/loaded in the order: \( A001, A101, A0, \Delta t2, \Delta t2, \Delta t1, \text{optDelta0, optDelta2, optDelta1, opt0, A1} \). This traversal yields (taking into account the in-place mapping) \( \text{MemorySize} = (M + m + 1)(N + n + 1) + 2(M - m + 1)(N - n + 1) = 1203 \) for the chosen values of the parameters \( M = N = 20, m = n = 4 \). The equivalent source code compatible with the proposed ordering is shown in Fig. 8. Only two partial constraints are essential to guide the subsequent scheduling stage: the basic set \text{optDelta0} must be produced after \( \Delta t1 \), and the input basic set \( A1 \) must be loaded after the production of \text{opt0}.

The increase of the granularity level entails a data-flow analysis with smaller groups of signals. This provides lower values for the memory size at the expense of the gradual expansion of the loop organization. At the same time, more constraints are conveyed to the scheduler. The memory size evaluation for the data-flow graph corresponding to granularity level 1 yields a storage requirement of 660 locations. The variance of the memory during the graph traversals is displayed in Fig. 7 for the DFG's of granularity levels 0 and 1. Similar evaluations for granularity levels 2, 3, and 4—the scalar level—yields the same memory size, i.e., 627 locations. The analysis of the code in Fig. 1 shows that \( (M + m + 1)(N + n + 1) + 2 = 627 \) locations represents indeed the absolute lower-bound on storage.

This example also shows that it is not necessary to descend to the scalar level in order to obtain a significant reduction for the memory size. Furthermore, even more crude evaluations for lower granularity levels (as that one corresponding to level 1) may be considered good enough for this stage of the synthesis. As the amount of constraints is relatively reduced, there is more freedom for scheduling and data-path allocation. Further memory adjustments can be done afterwards, taking into account the detailed scheduling decisions.

VI. OVERVIEW OF MAIN RESULTS

The implementation of the presented approach was done in \texttt{C++}, under the CATHEDRAL framework. It was tested on an HP 9000/735 workstation. The novel estimation method has been evaluated on three RMSP applications: 1) a singular value decomposition (SVD) updating algorithm [21]—an important algebraic kernel used e.g., in beamforming and Kalman filtering, 2) a motion detection algorithm [7] from a video coding application (see Fig. 1 for part of the code), 3) the kernel of a complex voice coding algorithm—essential component of a mobile radio terminal [32].

Table I shows the results obtained by the presented approach concerning the storage size, in comparison with those yielded by \texttt{s2p/agora}—the memory tool currently employed by the CATHEDRAL system—which uses a scalar-oriented technique for the in-place mapping. The results are provided for the different levels of granularity (column 2), up to the maximal loop depth—for which the number of signal instances (scalars) in the algorithm (column 3) equals the number of basic sets (column 4). The CPU times and the estimated storage requirements are listed in columns 5 and 6. The minimum number of memory locations \( (N) \) yielded by \texttt{s2p/agora} on the same algorithm representations are displayed in column 7. The last two columns show the absolute lower bounds on memory locations, and the relative difference between our estimations and these values.

Different from our approach, the memory tool \texttt{s2p/agora} interprets \textit{procedurally} the source code of the signal processing
algorithm. Employing a “left-edge” algorithm, s2p/agora finds the minimum number of memory locations only for one single operation ordering—provided by the code of the application. As our tool interprets the source code nonprocedurally, it can find better execution orderings (in terms of memory), as it happens for the SVD updating. This explains why our estimations from column 6 may be lower than the exact memory values obtained by s2p/agora (column 7) for the nonoptimized procedural specification.

It can be noticed from Table I that the relative difference between our estimations and the memory lower-bounds is decreasing towards 0 as the data-flow analysis becomes finer.\(^\text{15}\)

However, the CPU time is growing exponentially as getting closer to the scalar level. The results have been presented for all the granularity levels only to highlight the hierarchical capabilities of our model, and not for practical reasons. As it can be seen from the results, descending towards the scalar level is unnecessary and time consuming. Furthermore, this is impossible for most of the real-life RMSP applications, as shown in the sequel.

Table II exemplifies the use of our approach when the amount of scalars in the application is huge. s2p/agora—the memory tool in the CATHEDRAL system—cannot handle the motion detection algorithm for the sets of parameters shown in Table II. On the contrary, our approach can still handle these cases, although descending towards the scalar level is not possible any more. For the first set of parameters, the highest reachable granularity level was 2 (rather than 4). For the second set of parameters, the motion detection algorithm contains almost 14 million scalars. Only the first granularity level could be reached, but the relative difference between our estimation and the computed memory lowerbound is very acceptable.

Table III shows the area estimation for the same applications, with granularity level equal to 1, for different cycle budgets allocated for read/write operations. According to the layout model presented in [23], the actual silicon area occupied by the background memory is

\[
A = \text{Technology Factor} \cdot \text{bits} \cdot (1 + \alpha \text{Ports}) \cdot (N + \beta) \cdot [1 + 0.25(\text{Ports} + \text{Ports}_{sp} - 2)]
\]

The evaluation of the port requirements (number and type) is achieved by a partial ordering of the read/write operations within a given cycle budget, taking into account also the constraints derived from the DFG structure and traversal. All the signals are assumed to share the same global memory, having the wordlength equal to the maximum signal bit size.\(^\text{16}\)

The characteristics of the resulting memory are the wordlength, number of locations, and port distribution. The corresponding area is expressed in mm\(^2\), assuming a CMOS technology with 1.2 μm minimum geometry. The first area column in Table III displays the area values obtained from the layout model [23] embedded in our tool. The second area column displays the values obtained building the memory floorplan with a module generator.\(^\text{17}\) The results obtained by our method fit within a range of 10% the experimental results, confirming the validity of the layout model we use.

VII. CONCLUSIONS

In the context of background memory allocation for multidimensional signal processing circuits, we have addressed the problem of memory area evaluation for high-level non-procedural specifications of RMSP algorithms. In the paper, we have presented a novel technique founded on data-flow analysis, which allows to address this problem while operation scheduling is still unknown. In most multidimensional signal processing examples, the number of signal instances is huge, so a flattened data-flow analysis is impossible. Even if the size of the data-flow graph had allowed the flattening, the resulting

\(^\text{15}\) Usually, the estimations are crude for coarse-grain partitioning (granularity = 0). The reason is that the size of the basic sets is high, and there is insufficient information—relative to that granularity—concerning the production of the scalars within the basic sets. Consequently, the upper-bounds computed in Section V-B have to be conservative.

\(^\text{16}\) We have recently developed also a distributed memory allocation approach [4] based on the same mathematical concepts and data-flow view.

\(^\text{17}\) Currently, this module generator can build only RAMs with one R/W port.
controller would likely be too complex. Consequently, a new data-flow model grouping scalar signals in so-called basic sets is proposed. Based on this, a novel approach for assessing the memory area has been presented. Actually, the method incorporates a way to tradeoff the memory size with the complexity of both the computational effort and the controller to be achieved. These claims are substantiated with results for realistic applications extracted from video coding and mobile radio systems.

ACKNOWLEDGMENT

The authors would like to acknowledge their colleagues F. Franssen, M. van Swaaij, and L. Nachtegaele for the interesting discussions and permanent encouragement. They would also like to acknowledge their colleagues from IRISA (France) and D. Wilde (Univ. Oregon).

REFERENCES


Florent Balasa received the M.Sc. degree in computer science from the Polytechnic Institute of Bucharest in 1981, and the M.Sc. degree in applied mathematics from the University of Bucharest, Romania, in 1990.

He has worked over six years as a Senior Researcher at the Research Institute for Electronic Components, Bucharest. He has been a Ph.D. degree candidate at the Interuniversity Microelectronics Center (Katholieke Universiteit Leuven), Belgium since 1991. His research interests include high-level synthesis with emphasis of memory management issues, parallel processing, and data and control dependence analysis.
Francky Cathoor (S’86–M’87) was born in Temse, Belgium, in 1959. He received the Eng. and Ph.D. degrees in electrical engineering from the Katholieke Universiteit Leuven, Belgium, in 1982 and 1987, respectively.

From September 1983 to June 1987, he was a Researcher in the area of VLSI design methodologies for Digital Signal Processing, with Professor H. De Man and Professor J. Vandewalle as Ph.D. thesis advisors. First, at the EE Department of the Catholic University Leuven until the end of 1984; and afterwards at the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium. Since 1987, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, all within the VLSI system design methodology (VSDM) division at IMEC. His current research activities are mainly in the field of architecture design methods and system-level optimizing transformations for area and power, mainly oriented towards memory management and global communication optimization. The major target application domains are real-time signal and data processing algorithms in image, video and end-user telecom applications, and data structure dominated modules in telecom networks.

Dr. Cathoor has authored or coauthored about 130 papers, of which 2 received a Best Paper Award in the above-mentioned fields. In 1986, he received the Young Scientist Award from the Marconi International Fellowship.

Hugo De Man (M’81–SM’81–F’86) was born in Belgium on September 19, 1940. He received the Elec. Eng. degree and the Ph.D. degree in applied sciences from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1964 and 1968, respectively.

In 1968 he became a Member of Staff of the Laboratory for Physics and Electronics of Semiconductors at the University of Leuven, working on device physics and integrated circuit technology. From 1969 to 1971 he was at the Electronic Research Laboratory, University of California, Berkeley, as an ESRO-NASA Postdoctoral Research Fellow, working on computer-aided device and circuit design. In 1971, he returned to the University of Leuven as a Research Associate of the NFWO (Belgian National Science Foundation). In 1974 he became a Professor at the University of Leuven. During the winter quarter of 1974–1975, he was a Visiting Associate Professor at the University of California, Berkeley. He was an Associate Editor for the IEEE Journal of Solid-State Circuits from 1975–1980, and was European Associate Editor for the IEEE Transactions on Computer-Aided Design from 1982 to 1985. Since 1984 he has been Vice-President of the VLSI systems design group of IMEC (Leuven, Belgium), where the actual field of this research division is design methodologies for Integrated Systems for telecommunication. Examples are spread spectrum and ATM components design. Research of this group has been at the basis of the EDC-Mentor Graphics DSP-Station.

Dr. De Man is a corresponding member of the Royal Academy of Sciences, Belgium, and a member of the Royal Flemish Engineering Society (KIViV). He received the Best Paper Award at the ISSCC of 1973 on Bipolar Device Simulation and at the 1981 ESSCIRC Conference for work on an integrated CAD system. In 1986 he received, together with L. Claesen, the Best Paper Award in CAD from the ICCD-86 Conference, and in 1987 for best publication in the International Journal of Circuit Theory and Applications. In 1989, he received the Best Paper Award at the Design Automation Conference.