Abstract – In real-time multimedia processing systems a very large part of the power consumption is due to the data storage and data transfer. Moreover, the area cost is often largely dominated by memories. Hence, the optimization of the memory architecture is a crucial step in the design methodology for this type of applications. In deriving an optimized memory architecture, memory size computation is an important step in the data transfer and storage exploration stage. This paper investigates non-scalar methods for computing exactly the memory size in real-time multimedia algorithms. The approach is based on novel algebraic techniques specific to the data-flow analysis used in modern compilers. In contrast with previous works which utilize only approximate methods due to the size of the problems (in terms of number of scalars), this research aims to obtain exact determinations even for large applications.

1 Introduction

In real-time multimedia processing systems – including video an image processing, medical imaging, artificial vision, real-time 3D rendering, advanced audio and speech coding – a very large part of the power consumption is due to the data storage and data transfer. A typical system architecture includes custom hardware (application-specific accelerator datapaths and logic), programmable hardware (DSP core and controller), and a distributed memory organization which is usually expensive in terms of power and area cost. Data transfer and memory access operations typically consume more power than a datapath operation. For instance, fetching an operand from an off-chip memory for an addition consumes 33 times more power than the actual computation; even a transfer from an on-chip memory consumes about 4 to 10 times more power than the addition itself [2]. Moreover, the area cost is often largely dominated by memories. Hence, the optimization of the memory architecture is a crucial step in the design methodology for this type of applications. In deriving an optimized memory architecture, memory size estimation/computation is an important step in the data transfer and storage exploration stage. This problem has been tackled in the past both in register-transfer programs at scalar level [6, 4, 13, 11] and in behavioral specifications at non-scalar level [1, 15, 5, 16]. Good overviews of these techniques can be found in [2, 10].

This paper investigates non-scalar methods for computing exactly the memory size in real-time multimedia algorithms. This approach uses novel algebraic techniques specific to the data-flow analysis used in modern compilers [8]. In contrast with previous works which utilize only approximate methods due to the size of the problems (in terms of number of scalars), this approach aims to obtain exact determinations even for applications significantly large. Moreover, data-flow analysis enables the study of memory management tasks at the desired level of granularity – between whole array and the scalar level – trading-off computational effort, solution accuracy and optimality. The memory computation approach described in Sections 2 and 3 is intended to handle the entire class of “affine” specifications (therefore, a large class of real-time multimedia applications).

2 Computation of array reference size using algebraic transformations

In order to address the computation of the memory size necessary for the execution of a multidimensional signal processing algorithm, a simpler problem must be addressed first: the computation of the number of distinct scalars covered by a single array reference.

Example for \( (i = 0; ~ i \leq 511; ~ i + +) \)
\[ \text{for } (j = 0; ~ j \leq 511; ~ j + +) \]
\[ \text{for } (k = 0; ~ k \leq 511; ~ k + +) \]
\[ \cdots M[i+k][j+k] \cdots \]

How many memory locations are necessary to store the array reference \( M[i+k][j+k] \)? In spite of its apparent simplicity the problem is difficult. Moreover, it did not receive too much attention although it is the cornerstone of the exact memory computation. The correct answer for the example above is not the total number of iterator triplets \((i,j,k)\), that is \(512^3 \approx 134,217,728\). The reason is that the same scalar signal is obtained for different combinations of the iterators \(i, j, k\). For instance, the iterator vectors \((i,j,k) = (0, 1, 1)\) and \((i,j,k) = (1, 2, 0)\) yield the same scalar \(M[1][2]\). The answer is not \(1024^3 \approx 1,046,529\) either \((0 \leq i + k, j + k \leq 1022)\) since, e.g., the scalar \(M[0][512]\) is not addressed by any iterator vector \((i,j,k)\).

It must be emphasized that enumerative techniques can always be applied to compute the number of scalars in an array reference. These approaches are obviously simple and extremely efficient for array references with “small” iterator sets. However, in image and video processing applications most of the array references are characterized by huge sets of iterators: an enumerative technique, although very simple, will be too computationally expensive to use in such applications. For the illustrative example shown above, an enumerative algorithm was initially used for testing purpose, but the computation had to be eventually stopped. For such examples, algebraic techniques are the only hope.
A polyhedron is a set of points \( P \subset \mathbb{R}^n \) satisfying a finite set of linear inequalities: \( P = \{ x \in \mathbb{R}^n \mid A \cdot x \geq b \} \), where \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \). If \( P \) is a bounded set, then \( P \) is called a polytope. If \( x \in \mathbb{Z}^n \), then \( P \) is called an integral polyhedron/polytope. The set \( \{ y \in \mathbb{R}^m \mid y = A x , x \in \mathbb{Z}^n \} \) is called the lattice generated by the columns of matrix \( A \).

Each array reference \( M[\{i_1, \ldots, i_n\}] \cdots \{i_m, \ldots, \}} \) of an \( m \)-dimensional signal \( M \), in the scope of a nest of \( n \) loops having the iterators \( i_1, \ldots, i_n \) is characterized by an iterator space and an index space. The iterator space signifies the set of all iterator vectors \( i = (i_1, \ldots, i_n) \in \mathbb{Z}^n \) in the scope of the array reference. The index space is the set of all index vectors \( x = (x_1, \ldots, x_m) \in \mathbb{R}^m \) of the array reference.

If the loop boundaries are affine mappings with integer coefficients of the surrounding loop iterators, the increment steps of the loops\(^1\) are \( \pm 1 \), and the conditions in the scope of the array reference are relational and/or logical operations between affine mappings of the loop iterators\(^2\), then the iterator space can be represented by one or several disjoint integral (iterator) polytopes \( A \cdot i \geq b \), where \( A \in \mathbb{Z}^{(2n+e) \times n} \) and \( b \in \mathbb{Z}^{2n+e} \). The first \( 2n \) linear inequalities are derived from the loop boundaries and the last \( e \) inequalities are derived from the control-flow conditions (this representation is usually not minimal). The size of the iterator space is \( \text{Card} \{ i \in \mathbb{Z}^n \mid A \cdot i \geq b \} \).

If, in addition, the indices of an array reference are affine mappings with integer coefficients of the loop iterators, the index space consists of one or several linearly bounded\(^3\) lattices (LBL)\(^4\) – the image of a vector affine function over the iterator polytopes:

\[
\{ x = T \cdot i + u \mid A \cdot i \geq b , i \in \mathbb{Z}^n \}
\]

where \( x \in \mathbb{Z}^n \) is the index (coordinate) vector of the iterator space. The affine function is characterized by \( T \in \mathbb{Z}^{n \times m} \) and \( u \in \mathbb{Z}^m \). The size of the index space of an array reference is, therefore, \( \text{Card} \{ x = T \cdot i + u \in \mathbb{Z}^n \mid A \cdot i \geq b , i \in \mathbb{Z}^n \} \). The number of scalars addressed by the array reference is obviously the size of its index space.

Example The index space of the operand \( M[i+k][j+k] \) in the example above is the set

\[
\left\{ \begin{array}{c}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 1
\end{bmatrix} \begin{bmatrix}
i \\
j
\end{bmatrix} + \begin{bmatrix}
0 \\
0
\end{bmatrix}
\end{array} \right.
\]

where \( 0 \leq i,j,k \leq 511 \), \( i,j,k \in \mathbb{Z} \).

In general, the index space may be a set of linearly bounded lattices. E.g., a conditional instruction \( i \neq j \) determines two LBL’s for the array references within the scope of the condition – one corresponding to \( i \geq j + 1 \), and another corresponding to \( i \leq j - 1 \). As the iterator space of an array reference can be decomposed into a set of disjoint polytopes and

\(^1\)The loops having increment steps different from \( \pm 1 \) can be easily "normalized" with the affine transformation \( i = i' \cdot \text{Step} + \text{lower bound} \).

\(^2\)The polyhedral representation of the iterator space is still valid if the array reference scope also contains data-dependent (but iterator independent) conditions.

\(^3\)As the definition domain of the mappings is not \( \mathbb{Z}^n \) but a polytope – which is bounded by hyperplanes (characterized by linear equations).
polytope is empty or not (proper projection!).

**Example (cont’d)** The transformed iterator polytope is \( \{ j \in \mathbb{Z}^3 \mid \mathbf{A}' \cdot j \geq b \} = \{ 511 \geq j_1 - j_3 \geq 0, 511 \geq j_2 - j_3 \geq 0, 511 \geq j_3 \geq 0 \} \). As \( r = \text{rank } \mathbf{T}' = 2 \), its projection can be obtained eliminating the last iterator \( j_3 \) (as \( n - r = 1 \)) from the inequalities above. It follows that \( pr_2(\mathbf{A}' \cdot j \geq b) = \{ (j_1, j_2) \in \mathbb{Z}^2 \mid 511 \geq j_1 - j_2 \geq -511, 1022 \geq j_1 \geq 0, 1022 \geq j_2 \geq 0 \} \), which contains 784,897 points \( (j_1, j_2) \), since it can be verified that all these points \((j_1, j_2)\) are proper projections of iterator triplets \((j_1, j_2, j_3)\) in the iterator space \( \{ 511 \geq j_1 - j_3 \geq 0, 511 \geq j_2 - j_3 \geq 0, 511 \geq j_3 \geq 0 \} \). For instance, the point \((j_1, j_2) = (1, 2)\) in \( pr_2(\mathbf{A}' \cdot j \geq b) \) corresponds the non-empty 1-dimensional polytope \( \{ j_3 \in \mathbb{Z} \mid 1 \geq j_3 \geq 0 \} \). It follows that the size of the index space \( M[i + k][j + k] \) and, consequently, the necessary memory to store it, is 784,897 – therefore less than 0.6% of the number of the iterator triplets \((i, j, k)\).

### 3 Memory size computation using data-dependence analysis

Once the collections of array references for each multidimensional signal are extracted from the signal processing algorithm, an analytical partitioning into disjoint signal groups is performed for each collection. The aim of the partitioning process is to determine which parts of an array operand are not needed any more after the computation of the signals in a resulting array reference, 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.

The analytical partitioning of the array references in disjoint groups of signals can be performed by recursively intersecting the linearly bounded lattices (LBL’s) representing the index of the multidimensional signals in the algorithm. Let \( \{ x = T_1 i_1 + u_1 \mid A_1 i_1 \geq b_1 \} \), \( \{ x = T_2 i_2 + u_2 \mid A_2 i_2 \geq b_2 \} \) 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\(^4\) \( T_1 i_1 - T_2 i_2 = u_2 - u_1 \) having the elements of \( i_1 \) and \( i_2 \) as unknowns. If the system has no solution, the intersection is empty. Otherwise, let

\[
\begin{bmatrix}
i_1 \\
i_2
\end{bmatrix} = \begin{bmatrix} V_1 & V_2 \end{bmatrix} i + \begin{bmatrix} v_1 \\
v_2
\end{bmatrix}
\]

be the solution of the Diophantine system. Then the intersection

\[\text{is a new LBL defined by}
\{ x = T_1 V_1 \cdot i + T_1 V_2 \cdot u_1 \mid \begin{bmatrix} A_1 V_1 \\ A_2 V_2 \end{bmatrix} \cdot i \geq \begin{bmatrix} b_1 - A_1 V_1 \\ b_2 - A_2 V_2 \end{bmatrix} \} \]

After the partitioning of the array references, a polyhedral data-dependence graph with exact dependence relations can be produced (Fig. 2). The nodes in the graph correspond to the groups of signals determined analytically as shown above. The nodes in the data-dependence graph are weighted with the number of scalar signals in the group (computed as explained in Section 2). The arcs are weighted with the exact number of dependences between the groups of scalar signals corresponding to the nodes. The data-dependence graphs can be built at the desired level of granularity, depending on the loop nesting level. Our concept of granularity is somewhat similar to the fine-grained and coarse-grained models used in [9].

**Example** The graph derived from this code is shown in Fig. 2:

\[
\text{optDelta}[0] = 0;
\text{for } (i = 4; i \leq 20; i + + )
\text{for } (j = 4; j \leq 20; j + + )
\text{Delta}[i][j][0] = 0;
\text{for } (k = i - 4; k \leq i + 4; k + + )
\text{for } (l = j - 4; l \leq j + 4; l + + )
\text{Delta}[i][j][9*(k-i)+l-j+41] = A[i][j]-A[k][l]@1 + \text{Delta}[i][j][9*(k-i)+l-j+40];
\text{optDelta}[17*i + j - 72] = Delta[i][j][81] + \text{optDelta}[17*i + j - 72];
\text{opt} = \text{optDelta}[289];
\]

These polyhedral dependence graphs allow the computation of the memory size by performing a relative lifetime analysis, without simulating the computation of the code and without decomposing the arrays into scalars (which is beneficial, as in most applications the number of scalars is extremely large). When the algorithmic specification is non-procedural – the computation ordering being basically free, constrained only by dependence relations – these graphs can be used to estimate effi-
ciently the memory size [1]. When the specifications are procedural, the polyhedral dependence graphs can be used to compute exactly the memory size necessary for the algorithm execution. In the example above the memory size is exactly 627 locations.

4 Results

The implementation of the presented approach was done in C++ and was tested on a Sun Blade 100 workstation. The evaluation of the novel technique has been carried out on several multidimensional signal processing applications like, for instance: (1) A singular value decomposition (SVD) updating algorithm [7] for use in spatial division multiplex access (SDMA) modulation in mobile communication receivers. The SVD updating is an important algebraic kernel used, for instance, in beamforming, recursive least squares estimation, and Kalman filtering. For mobile telephony, signals on the same carrier frequency but arriving from different locations can be separated by means of antenna arrays and advanced computational techniques (as, e.g., SVD updating), which allows to enlarge significantly the network capacity. (2) A motion detection algorithm for use in the transmission of real-time video signals on data networks. The video signals must be compressed by video-coders/decoders to fit in the (limited) bandwidth of the network. An approximate but effective algorithm uses the idea of motion compensation – the motion estimation of part of the image between consecutive frames. (3) The kernel of a voice coding application – essential component of a mobile radio terminal.

Table 1 summarizes the results of our experiments. The computation times are quite large due to the exponential complexity of some of the techniques employed (e.g., the Fourier-Motzkin elimination [3]). However, a scalar approach based on computation simulation took several hours to complete.

5 Conclusions and future work

This paper has presented a non-scalar model for computing the memory size required in real-time multimedia algorithms, where the storage of large multidimensional signals causes a significant cost in terms of both area and power consumption. The approach is based on novel algebraic techniques specific to the data-flow analysis used in modern compilers. In order to handle high throughput applications, the design space exploration system will be extended with a preprocessing stage – exploiting the (initially unknown) parallelism existent in the source code of the specification.

In addition, this model will be used in the synthesis of a multilevel memory architecture optimized for area and/or power, subject to timing constraints. The synthesis of an optimal memory hierarchy will comprise two subtasks: a data reuse exploitation – which will employ data-flow analysis techniques to exploit the temporal locality of data, deciding which intermediate copies have to be made for accessing the data in a power-efficient way, followed by the proper memory allocation and assignment step – which will decide the distributed (hierarchical) memory architecture.

References


<table>
<thead>
<tr>
<th>Application</th>
<th>Signals</th>
<th>Memory</th>
<th>CPU [m:s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVD updating</td>
<td>25,717</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motion detection</td>
<td>72,543</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vocoder kernel</td>
<td>28,559</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Results for signal processing applications