EFFICIENT ASSIGNMENT ALGORITHM FOR MAPPING MULTIDIMENSIONAL SIGNALS INTO THE PHYSICAL MEMORY*

Ilie I. Luican* Hongwei Zhu† Florin Balasa‡

* Dept. of Computer Science, University of Illinois at Chicago, Chicago, IL
† ARM, Inc., Sunnyvale, CA
‡ Dept. of Computer Science, Southern Utah University, Cedar City, UT

ABSTRACT

The storage requirements in data-intensive multidimensional signal processing systems have a significant impact on the system performance as well as on essential design parameters, like the overall power consumption and chip area. This paper addresses the problem of efficiently mapping the multidimensional signals from the algorithmic specification of the system into the physical memory. Different from all the previous mapping models that aim to optimize the memory sharing between the elements of a same array, this proposed assignment algorithm takes also into account the possibility of memory sharing between different arrays. As a consequence, the experiments with this novel signal-to-memory mapping approach exhibit important savings of data storage resulted after mapping.

Index terms– Memory management, signal-to-memory mapping, multidimensional signal processing, storage requirement

1. INTRODUCTION

The multidimensional signal processing applications are algorithmically specified in high-level programming languages, the main data structures being multidimensional arrays. Since these (typically large) arrays must be stored during the execution of the application code, an important memory management problem is the *mapping* of the arrays into the physical memory [1]. The goals of this operation are the following:

(1) to map the arrays from the behavioral specification into an amount of data storage as small as possible;

(2) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity;

(3) to ascertain that any distinct scalar signals (array elements) *simultaneously alive* are mapped to distinct storage locations. The lifetime of a scalar signal is the time interval between the clock cycles when the scalar is *produced* or written, and when is read for the last time, i.e. *consumed*, during the code execution. Two scalars are simultaneously alive if their lifetimes do overlap. Obviously, in such a case, they must occupy different memory locations; otherwise, they can share the same location.

Several mapping models have been proposed in the past, trading off between the first two goals (that is, accepting a certain excess of storage to ensure a less complex address generation hardware) while ascertaining that the third goal is strictly satisfied.

De Greef *et al.* analyze the canonical linearizations[1] of the arrays in the algorithmic specification and, for each linearization, the largest distance between two simultaneously alive array elements is computed [2]. The minimum (over all the canonical linearizations of the array) largest distance plus 1 is the size of a storage window where the array can be mapped into the data memory without any conflict (i.e., simultaneously alive elements are stored in distinct locations). The linearization yielding the minimum largest distance is finally selected. The values of the mapping function are the positions of the array elements in the selected linearization, followed by a *modulo* operation (whose operand is the corresponding distance plus 1) that wraps the set of "virtual" memory locations into a smaller set of actual physical locations. Since the number of analyzed linearizations ($2^m \cdot m!$) increases fast with the signal dimension m, the computation times implied by this mapping model can be significant.

In order to avoid the inconvenience of analyzing different linearization schemes, Tronçon *et al.* proposed to compute an m-dimensional bounding box (window) in the original m-dimensional index space of the array [3]. The elements of this window are computed separately, and they are the largest index differences between simultaneously alive elements in each dimension, plus 1. If this mapping window of an m-dimensional array A is $W = (w_1, \ldots, w_m)$, the window elements w_i are used as operands in modulo operations that redirect all accesses to the array. For instance, an access to the element $A[index_1] \ldots [index_m]$ is redirected to $A[index_1 \bmod w_1] \ldots [index_m \bmod w_m]$ (relative to a base address in the data memory). The way the window elements are computed ensures that any two distinct array elements simultaneously alive cannot be redirected to the same storage location.

Lefebvre and Feautrier, addressing parallelization of static control programs, developed in [4] an approach based on modular mapping, as well. They first compute the lexicographically maximal "time delay" between the write and the last read operations,

*This research was sponsored by the U.S. National Science Foundation (DAP 0133318).

[1] The row and, respectively, the column concatenations of a 2-D array are canonical linearizations.

which is a super-approximation of the distance between conflicting index vectors [5] (i.e., whose corresponding array elements are simultaneously alive).

Darte *et al.* proposed a very refined mathematical framework, establishing a correspondence between valid linear storage allocations and integer lattices called *strictly admissible* relative to the set of differences of the conflicting indices [5]. Heuristic techniques for building strictly admissible integer lattices (hence, building valid storage allocations) are proposed.

Luican *et al.* proposed a technique based on lattice decomposition which can be used to implement both mapping approaches [2] and [3], while being significantly faster [6]. In addition, this technique can be extended to deal with signal mapping into a hierarchical memory organization.

These signal-to-memory mapping approaches treat *separately* the arrays from the algorithmic specification, computing windows in the physical memory for each individual array. They exploit the possibility of memory sharing between the elements of a same array. However, since the arrays are handled separately, the possibility of memory sharing between elements of different arrays is inherently ignored. This can lead to an excessive data storage, as Section 2 will illustrate. Interestingly, the possibility of memory sharing between elements of different arrays with disjoint lifetimes was observed long time ago [1] and it has been taken into account by several approaches for memory size evaluation (e.g., [7], [8]). However, the inter-array memory sharing is more difficult to achieve during signal-to-memory assignment, since an explicit correspondence between the array elements and their addresses in the physical memory must be indicated.

This paper presents a novel signal-to-memory assignment algorithm that can take into account the possibility of memory sharing between elements of different arrays, even when the arrays do not have disjoint lifetimes. The rest of the paper is organized as follows. Section 2 discusses an illustrative example, analyzing the results of two past mapping techniques and explaining the motivation of this research. Section 3 presents the basic ideas of this novel assignment algorithm. Section 4 addresses implementation aspects and discusses the experimental results. Section 5 summarizes the main conclusions of this work.

2. DISCUSSION ON AN ILLUSTRATIVE EXAMPLE

In the illustrative example in Fig. 1, the A-elements produced in the first loop nest are consumed in the second loop nest; the B-elements produced in the second loop nest are consumed in the third loop nest; the C-elements are produced and consumed in the third loop nest. The variation of the storage requirement as a function of the number of executed assignments is shown in Fig. 2. If we handle the three arrays independent of each other, the storage requirement for A is 16 memory locations, for B is 25 locations, whereas C needs 18 locations (since at most 18 C-elements are simultaneously alive). Assuming that the three arrays are stored in separate windows of the physical memory, the minimum data memory for the whole code in Fig. 1 is 16+25+18=59 locations. Otherwise, since the three arrays can share data memory due to the different lifetimes of their elements, the minimum data storage

```
int  A[7][4], B[9][5], C[11][6] ;
for ( i=0; i<=6; i++ )        // The first loop nest
  for ( j=0; j<=3; j++ )
    if ( 3<=i+j  &&  i+j<=6 )   A[i][j] = 16 ;
for ( i=0; i<=8; i++ )        // The second loop nest
  for ( j=0; j<=4; j++ )
    if ( 4<=i+j  &&  i+j<=8 )
      if ( i<=3 )  B[i][j] = A[i][j-1] + A[6-i][4-j] ;
      else  A[i][j] = 32 ;
for ( i=0; i<=13; i++ )       // The third loop nest
  for ( j=0; j<=5; j++ )
  {  if ( 5<=i+j  &&  i+j<=10 )
       if ( i<=4 )  C[i][j] = B[i][j-1] + B[8-i][5-j] ;
       else  C[i][j] = 64 ;
     if ( 8<=i+j  &&  i+j<=13 )   ... = C[i-3][j] ;
  }
```

Figure 1: Illustrative example.

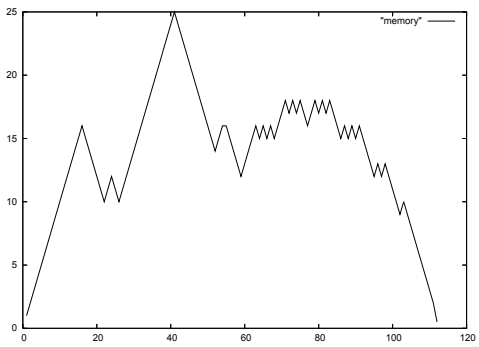

Figure 2: Memory trace [8] for the illustrative example.

ensuring the code execution is only 25 locations, as shown by the graph in Fig. 2.

The minimum data storage (of 25 locations for this example) represents the *tight lower bound* for which the execution of the code is possible. However, in practice, this amount of storage is difficult to reach (although still possible!) since it would require a complex control and hardware for address generation. Instead, the designers apply more regular signal-to-memory mapping techniques to compute the physical addresses in the data memory for the array elements in the application code. These mapping models actually trade-off an excess of storage for a less complex address generation hardware. The results given by two mapping techniques – briefly described in Section 1 – will be discussed below.

(1) The assignment model [2] analyzes the canonical linearizations of A, B, and C, measuring the maximum distance between the simultaneously alive elements, relative to the respective linearizations. For instance, the maximum distance for signal A is 21 (e.g., between the elements $A[0][3]$ and $A[6][0]$ in the linearization by row concatenation). Therefore, a linear window of 21+1=22 would suffice to ensure that the simultaneously alive elements are stored in distinct locations. Possible mappings are $A[i][j] \mapsto Mem_A[(4i+j) \bmod 22]$ or $A[i][j] \mapsto Mem_A[(7j+i) \bmod 22]$, where Mem_A is the address of a memory window assigned to signal A. Similarly, signal B needs a memory window of 37

locations. Indeed, the maximum distance between live elements is 36 (e.g., between the elements $B[4][0]$ and $B[4][4]$ in the linearization by column concatenation). Finally, signal C needs a memory window of 19 locations since the maximum distance between live C-elements is 18 (for instance, between the elements $C[3][4]$ and $C[6][4]$ – by row concatenation – in the iteration (i , j) = (6 , 4) of the third loop nest, before the consumption of $C[3][4]$). For this example, the model [2] yields a data memory of $|Mem_A| + |Mem_B| + |Mem_C|$=22+37+19=78 locations.

(2) The assignment model [3] computes 2-D windows for the signals A, B, and C, large enough along each dimension to bound their elements simultaneously alive. For signal C, the maximum distance for each index between live elements are $d_1 = 3$ and $d_2 = 5$ (since $C[3][4]$ and $C[6][4]$ are simultaneously alive, and also $C[5][0]$ and $C[5][5]$). The bounding window of C computed by the model is $W_C = (d_1 + 1, d_2 + 1) = (4, 6)$. Each array element is first mapped into this window: $C[i][j] \mapsto W_C[i \bmod 4][j \bmod 6]$; in its turn, the window is mapped into the physical memory by a typical canonical linearization (row or column concatenation). The overall storage requirement is $|Mem_A| + |Mem_B| + |Mem_C|$=28+66+24=118 locations.

Actually, the signals A and C in the illustrative example have disjoint lifetimes. Also, many B- and C-elements have disjoint lifetimes and can share same memory locations. The assignment models discussed above are unable to exploit this.

3. THE FLOW OF THE ASSIGNMENT ALGORITHM

The idea of the mapping algorithm is to search for a pairwise grouping of the arrays that is likely to yield the largest benefit in terms of data storage reduction by mutual memory sharing. Therefore, the elements of an array will be able to share the same storage locations only between themselves, or with the elements of one other array, but not to *all* the other arrays. This apparent limitation entails only a reasonable increase in the hardware cost for address generation (see the last paragraphs of the previous section), while attempting to maximize the benefit of memory sharing between distinct arrays. The flow of the algorithm is given below:

Step 1: For every array A in the algorithmic specification, compute the size of the memory window Mem_A, based on DeGreef's mapping model [2], that is, analyzing the canonical linearizations of the arrays and taking the smallest maximum distance between simultaneously alive elements. The initial memory windows are also evaluated using Tronçon's mapping model [3] and the minimum window provided by any of these two models is selected.

The implementation of this step uses the technique based on disjoint linearly bounded lattices described in [6]. This technique yields a much faster implementation than the original implementations of both mapping models [2] and [3].

Step 2: Build a complete graph G, where each vertex represents an array in the application code. Compute weights for every edge (A,B) in the following way:

(a) if the two arrays A and B have disjoint lifetimes, the weight is $\min\{size(Mem_A), size(Mem_B)\}$;

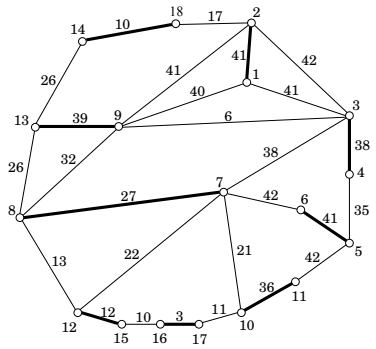

Figure 3: The maximum weighted matching (bold edges).

(b) when the lifetimes of the two arrays overlap, compute the maximum distance between the locations occupied by simultaneously alive A- and B-elements, taking into account the mapping functions found at *Step 1* and assuming the two memory windows are contiguous; the size of the common window Mem_{AB} is this maximum distance plus 1; the weight of the edge (A,B) is $size(Mem_A) + size(Mem_B) - size(Mem_{AB})$. This weight represents the data storage saved when the two arrays A and B share the same memory space versus the situation when the two arrays would be stored separately (in disjoint memory windows).

Step 3: Find the maximum weighted matching in the graph G. A matching in graph is a set of edges, no two of which meet at a common vertex. The weight of the matching is the sum of the weights of its edges. A maximum weighted matching represents a matching of maximum weight, as shown in Fig. 3 for a graph with 18 vertices. In our case, the graph is complete (there is an edge between every pair of vertices) and the matching will produce the most beneficial pairwise grouping of the arrays in terms of storage reduction. The matching solution will maximize the overall savings of data storage when the arrays are sharing pairwise the memory space. Note that even larger savings could be achieved, in principle, if more complex array groups shared the same memory space, but the computation effort would become prohibitive.

Maximum matching has been a subject of interest in graph theory for the last 50 years. Algorithms were first developed for matching on bipartite graphs. For non-bipartite graphs, most of the best matching algorithms are based on theorems proved by Berge in [9], who proposed searching for augmenting paths as a general strategy for maximum matching. Based on Berge's theorems, Edmonds proposed an efficient algorithm whose computation time is proportional to V^4, where V is the number of vertices [10]. The algorithm works by finding augmenting paths by a tree search combined with a process of shrinking certain subgraphs called *blossoms* into single nodes of a reduced graph (most often Edmond's algorithm is called the "blossom shrinking algorithm"). The fastest existing algorithm under the assumption of integral costs that are not particularly high was developed by Gabow and Tarjan [11]. Since the graphs built by this mapping algorithm have, typically, less than 100 vertices (the vertices being the arrays

Application (# Arrays)	# Array elements	Memory size [3]	Memory size [2]	**Memory size**
Motion detection (4)	318,367	9,525	9,636	**9,524**
Regularity detect. (6)	4,752	4,353	3,879	**2,817**
Gauss. blur filter (7)	177,167	48,646	50,448	**26,504**
SVD updating (9)	386,472	17,554	16,754	**10,360**
Voice coder (54)	33,619	13,104	13,224	**12,690**

Table 1: Experimental results.

in the application code), an older algorithm of cubic complexity due to Gabow [12] is sufficient for our benchmark tests, the computation times for finding the maximum weighted matching being under 1 second.

Step 4: Compute the overall data storage corresponding to the maximum weighted matching in the graph. Take also into account the possibility of disjoint lifetimes between entire arrays in this matching (in which case, the common windows of the two pairs can overlap). Determine the mapping functions for each array.

4. EXPERIMENTAL RESULTS

A software tool incorporating the algorithm described in this paper, performing the mapping to the data memory of the multidimensional arrays from a given algorithmic specification, has been implemented in C++. For the syntax of the algorithmic specifications, we adopted a subset of the C language (see, e.g., the code example in Fig. 1). Table 1 summarizes the results of our experiments. The benchmarks used are: (1) a motion detection algorithm used in the transmission of real-time video signals on data networks, (2) a real-time regularity detection algorithm used in robot vision, (3) a 2-D Gaussian blur filter from a medical image processing application which extracts contours from tomograph images in order to detect brain tumors, (4) a singular value decomposition (SVD) updating algorithm used in spatial division multiplex access (SDMA) modulation in mobile communication receivers, in beamforming, and Kalman filtering; (5) the kernel of a voice coding application, an essential component of a mobile radio terminal. Table 1 shows some basic characteristics of the benchmark codes: the numbers of arrays and the numbers of scalars (array elements); the last three columns display the data memory size *after mapping* obtained when employing the assignment models [3], [2], and this algorithm (bold fonts), respectively.

These experiments show that the novel mapping approach, exploiting the possibility of memory sharing between arrays, produces better results in terms of data storage than two previous techniques [2], [3] dealing with each array one by one. The amount of storage reduction relative to other previous techniques depends on the application, on how much the array lifetimes are overlapping with one another. For instance, in the case of the motion detection (for the set of parameters $M = N = 64, m = n = 4$), the results of both previous techniques are very good. On the other hand, for other benchmarks the reduction of data storage achieved by our technique is very significant. E.g., for the 2-D Gaussian blur filter, the storage reduction is over 46% when compared to both methods [2] and [3]; for the SVD updating the reduction is about 40%.

The computation times were not displayed in Table 1 due to lack of space: for the benchmarks in the table, the CPU times when running our algorithm were between tens of seconds and 2 minutes on a PC with a 1.85 GHz Athlon XP processor. The computation times for the mapping techniques [2] and [3] (with our implementation [6], which is much faster than the original ones) are lower; this is not unexpected since our algorithm uses those techniques in *Step 1*. Evaluating the capability of memory sharing between arrays consumes extra computation time, but this is a price worth being paid when the storage reduction is significant.

5. CONCLUSIONS

This paper has addressed the problem of assigning the multidimensional arrays from high-level algorithmic specifications of signal processing applications to the physical memory. Different from all the previous techniques that handle the arrays separately, this novel approach exploits the possibility of inter-array memory sharing, yielding significant savings of data storage.

References

[1] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Boston: Kluwer Academic Publishers, 1998.

[2] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on "Parallel Processing and Multimedia," A. Krikelis (ed.), in *Parallel Computing*, Elsevier, vol. 23, no. 12, pp. 1811-1837, Dec. 1997.

[3] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *Verification, Model Checking and Abstract Interpretation*, A. Coresi (ed.), pp. 167-181, 2002.

[4] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, pp. 649-671, 1998.

[5] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.

[6] I.I. Luican, H. Zhu, and F. Balasa, "Signal-to-memory mapping analysis for multimedia signal processing," *Proc. Asia & South-Pacific Design Automation Conf.*, Yokohama, Japan, 2007, pp. 486-491.

[7] P.G. Kjeldsberg, F. Catthoor, and E.J. Aas, "Data dependency size estimation for use in memory optimization," *IEEE Trans. Comp.-Aided Design of ICs and Syst.*, vol. 22, no. 7, pp. 908-921, 2003.

[8] F. Balasa, H. Zhu, and I.I. Luican, "Computation of storage requirements for multi-dimensional signal processing applications," *IEEE Trans. on VLSI Systems*, vol. 15, no. 4, pp. 447-460, April 2007.

[9] C. Berge, "Two theorems in graph theory," in *Proc. Nat. Acad. Science USA*, vol. 43, pp. 842-844, Sept. 1957.

[10] J. Edmonds, "Paths, trees, and flowers", in *Canadian J. of Mathematics*, vol. 17, pp. 449-467, 1965.

[11] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for general graph-matching problems," *J. of the ACM*, vol. 38, no. 4, pp. 815-853, 1991.

[12] H.N. Gabow, *Implementation of Algorithms for Maximum Matching on Non-Bipartite Graphs*, Ph.D. Thesis, Stanford University, 1973.