Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications[†]

Eddy De Greef, Francky Catthoor[*], Hugo De Man[*]

IMEC
Kapeldreef 75
B-3001 Heverlee
Belgium
e-mail: {degreef,catthoor,deman}@imec.be

Abstract

In this paper, we present some strategies that are capable of reducing the required memory sizes and power consumption for a large class of data-intensive multimedia applications. This class consists of static control programs with large multi-dimensional arrays and (piece-wise) affine storage and execution order. These strategies are equally well suited for parallel and mono-processing applications, and are particularly useful in an embedded application context, where memory size and power consumption usually are the main cost factors.

The main objective of these strategies is to reuse memory as much as possible by obtaining an optimal storage order for each of the arrays present in a program through (the equivalent of) data-transformations. Although size reduction is the main objective, an added benefit is the fact that the power consumption is also reduced due to the decreased capacitive load of the memories. The memory size reduction task is part of an overall memory size and power reduction methodology called ATOMIUM, in which other tasks can increase its effectiveness (e.g. through loop transformations), but it can also be used on a stand-alone base.

The presented strategies are based on an exact mathematical modeling of the memory occupation presented in an earlier paper. Here the most promising strategies are explored in more detail and their computational complexity is investigated. Their feasibility and effectiveness is demonstrated by experimental results for some real-life multimedia applications, for which a considerable size reduction was obtained.

[†]This research was sponsored by Siemens A.G., München, Germany

[*]Professor at the Katholieke Universiteit Leuven, Belgium.

1 Introduction

Two of the main cost factors in embedded multimedia applications are the chip area and the power consumption. Especially for data-intensive algorithms, between 50 and 80% of the chip area is occupied by memories [1]. Moreover, most of the power is consumed by memory accesses [1, 2, 3]. We have demonstrated this on several important multimedia applications including a H263 video decoder [4] and a motion estimation kernel [5]. Therefore the reduction of the background memory size is a crucial task in a multimedia system context. Due to the recent introduction of parallelism in the multimedia domain, optimization of memory related issues has become even more important.

In this paper we present novel techniques for reducing the memory *sizes* by reusing memory locations for arrays as much as possible through (the equivalent of) data-transformations. Our reduction strategy is based on an exact mathematical modeling of memory usage that we presented earlier [6]. The size reduction we obtain also influences the power consumption in a positive way (due to the decreased capacitive load during memory transfers), but here we mainly focus on size aspects. However, this task is part of our ATOMIUM data storage and transfer exploration environment [7]. Other tasks in this environment focus more on the power reduction issues (e.g. memory hierarchy decisions). Moreover the effectiveness of the memory size reduction task is heavily influenced by preceeding tasks (e.g. loop transformations), which enable memory reuse.

The strategies presented in this paper are applicable to a large class of parallel processing applications and architectures (SIMD, MIMD) with either shared and/or local memories, but are equally well suited for mono-processor applications. The strategy that we have selected for implementation consists of two major phases: in a first phase we optimize the internal storage order for each array separately, in order to increase the memory reuse between elements of the same array. In the second phase we exploit the remaining freedom to globally optimize the memory reuse between elements of the different arrays. In this paper we will mainly concentrate on the first phase, and only briefly discuss the second phase. The second phase is the main topic of a future paper [8]. In neither phase the memory access ordering is affected in any way, so these strategies do not interact with other issues of the parallel compilation process such as memory contention or real-time constraints, as only the address expressions are altered.

This paper is organized as follows: in section 2 we discuss some related work. In section 3, we briefly review our memory occupation models, followed by a discussion of our assumptions in section 4. Next, in section 5, we present several possible optimization strategies, and discuss the most promising one more in-depth in section 6. Finally, we present some experimental results in section 7, and draw some conclusions in section 8.

2 Related Work

In [9] the concept of using an *address* reference window in order to reduce the memory size requirements for multi-dimensional arrays was introduced. However, the presented method for calculating the window is not exact, as it provides only an upper bound. Moreover, no method is provided to optimize the storage order in order to minimize the windows. In this paper, we present an exact method to calculate the window and a method to obtain a good storage order.

In [10] and [11], the memory reuse problem is also discussed. These methods do not take into account the exact storage order of the arrays though, and only provide estimates on memory usage to steer other optimization tasks.

The problem of reducing memory size requirements is strongly related with the problem of optimizing data locality in (parallel) programs. In [12] a technique for calculating an approximate *index* reference window and a local memory data management strategy are presented. Also, in [13], an algorithm for performing both loop and data transformations to improve locality is presented. However, these locality improvement techniques are targeted towards improved performance and therefore concentrate on single loop nests. The problem of memory size minimization cannot be tackled by looking at individual loop nest though, and requires a strategy that takes a global view.

The problem of large storage requirements for single-assignment programs has also attracted the attention of other researchers. In [14] and [15], the principle of memory reuse through projection of multi-dimensional arrays is described. However, only a memory reuse analysis model is provided, together with a set of necessary and sufficient constraints that have to be satisfied, but no strategy for obtaining a good projection is presented. Moreover, memory reuse between different arrays is not considered, as in our approach.

3 Memory Occupation

In [6] we have presented some formal models describing in closed forms the exact memory occupation for each individual array element of single assignment static control programs with a (piece-wise) affine execution order and a (piece-wise) affine storage order. We will now briefly present these models and illustrate the concepts by means of a simple example.

```
     int A[3N][N];
     ...
     for ( i = 0; i < 2*N; ++i )
        for ( j = 0; j < N; ++j )
S1:        A[i][j] = f(...);
     ...
     for ( k = N; k < 3*N; ++k )
        for ( l = 0; l < N; ++l )
S2:        ... = g(A[k][l]);
     ...
```

First of all, we can describe the *iteration domains* (ID) of statements S1 and S2. Each integral point inside these domains corresponds to an *operation*, i.e. an execution of a statement. Similarly, we can describe the elements of array A by means of a *variable domain* (VD). Each integral point inside this domain corresponds to an element of A. The descriptions of ID_1, ID_2 and VD_A of our example are the following[1]:

$$\begin{aligned} ID_1 &= \{\ (i,j) \ \ | \ \ 0 \le i \le 2N-1,\ 0 \le j \le N-1\ \} \\ ID_2 &= \{\ (k,l) \ \ | \ \ N \le k \le 3N-1,\ 0 \le l \le N-1\ \} \\ VD_A &= \{\ (a_1,a_2) \ \ | \ \ 0 \le a_1 \le 3N-1,\ 0 \le a_2 \le N-1\} \end{aligned}$$

[1]Note that in this paper, we implicitly assume that all mathematical variables are integral.

Given a (piece-wise) affine execution order, we can calculate the execution date of every operation. Similarly, for a given (piece-wise) affine storage order of array `A` in a linearly addressed memory, we can calculate the memory addresses of each of the array elements. For the example, we assume the following time order functions: $t_{\texttt{S1}}(i,j) = iN + j$ and $t_{\texttt{S2}}(k,l) = C + kN + l$, where C is a constant, and storage order function: $a_A(a_1, a_2) = N * a_1 + a_2$.

Next, we can mathematically describe the elements of `A` that are being accessed by the different statements. Write and read accesses are described by means of *definition domains* (DD) and *operand domains* (OD) respectively. For the example, we have the following DD's and OD's[2]:

$$DD_{11A} = \{\ (a_1, a_2)\ \ |\ \ \exists i,j \texttt{ s.t. } a_1 = i,\ a_2 = j,\ 0 \le i \le 2N-1,\ 0 \le j \le N-1\ \}$$
$$OD_{21A} = \{\ (a_1, a_2)\ \ |\ \ \exists k,l \texttt{ s.t. } a_1 = k,\ a_2 = l,\ N \le k \le 3N-1,\ 0 \le l \le N-1\ \}$$

For a static control single-assignment program, we can detect the value-based flow dependences by intersecting the OD's and DD's of each array. In the example, there is a flow dependence from statement `S1` to statement `S2`. By intersecting DD_{11A} and OD_{21A}, we can find out what elements of `A` contribute to the flow dependence. We also know that we have to store these elements in memory from the moment that they have been produced by statement `S1` until *at least*[3] the moment they are being consumed by statement `S2`. So for each flow dependence, we can derive when the contributing elements are being produced and consumed (via the execution order of the corresponding iteration domains) and where they will be stored in memory (via the storage order of the array). Consequently, we can describe the set of address-time tuples being occupied by the flow dependence as follows[4]:

$$\begin{aligned} BOATD_{1121A} = \{\ (a,t)\ \ |\ \ & \exists\ i,j,k,l,a_1,a_2 \texttt{ s.t.} \\ & a = a_A(a_1,a_2),\ a_1 = i = k,\ a_2 = j = l, \\ & 0 \le i \le 2N-1,\ 0 \le j \le N-1, \\ & N \le k \le 3N-1,\ 0 \le l \le N-1, \\ & t_{\texttt{S1}}(i,j) \le t \le t_{\texttt{S2}}(k,l)\ \} \end{aligned}$$

This domain, which consists out of address-time tuples that are occupied due to a flow-dependence, is called a *binary occupied address-time domain* (BOATD). A graphical representation of the domains corresponding to the example is given in figure 1 (for the case where N equals 3).

If we know the BOATD for each dependence in the program, then we exactly know when each memory location is occupied. In general, different dependences of the *same* array can have overlapping BOATD's (i.e. when an array element contributes to several dependences). We will refer to the union of the different BOATD's of an array as the *occupied address-time domain* (OATD) of that array.

[2]The second subscript corresponds to the number of the definition or operand; for the example we have only 1 operand/definition per statement.

[3]At least, because other statements may consume the same elements later on too.

[4]Note that this equation can easily be further simplified.

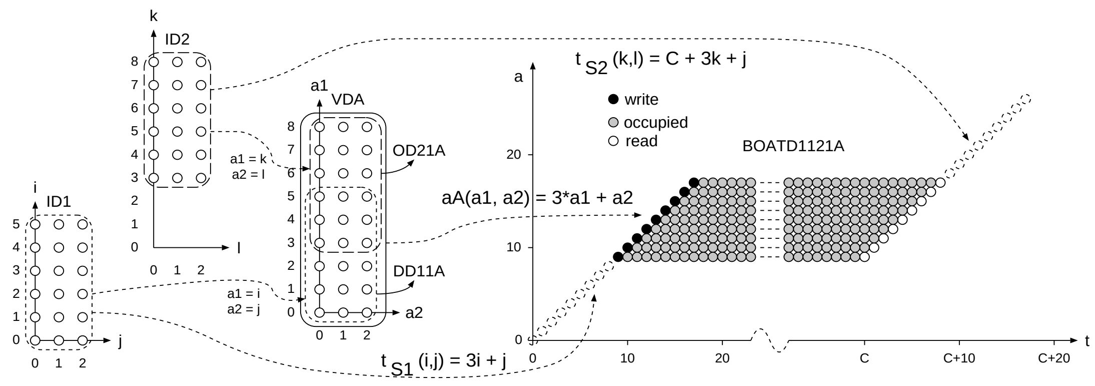

Figure 1: The different domains corresponding to the example.

In [6], we also describe a set of necessary and sufficient conditions that have to be satisfied by the execution order of the program and the storage order of each of the arrays in order to be valid. Due to space limitations, we only provide an intuitive summary here:

- Each array element should be produced before it is consumed.
- No array element should be written at a memory location that is being occupied by another array element at that time.

An ideal execution and storage order optimization strategy for area and power would have to take these conditions into account to arrive at a global optimum. Unfortunately, although these conditions may seem fairly obvious in textual form, their mathematical counterparts can be relatively complex and quite hard to evaluate in practice (especially the second set) and we therefore have to relax them to simpler (sufficient) ones in order to be able to obtain a good solution (see section 5).

4 Assumptions

In the sequel we assume that the *relative* execution order has been fixed, and that we have assigned each (part of an) array to a memory. The storage order of the arrays inside these memories on the other hand, is the subject of our optimizations. There is no direct interaction with other parallel compiler tasks, provided that the relative order of the memory accesses has been fixed already[5].

These are realistic assumptions, as we have applied this approach successfully in the past to obtain a memory-efficient SIMD-type architecture for implementing block-oriented video algorithms such as motion estimation [16]. An overview of the obtained architecture is shown in figure 2.

The architecture contains a few shared frame memory banks and several processing elements (PE's) with local memories. We first decided on a detailed data-distribution and a detailed data-transfer schedule, both for the transfers between the shared memory and the local memories and for the transfers between the local memories and the PE's.

[5]In the future we also intend to extend our techniques to take into account (hardware-controled) caches.

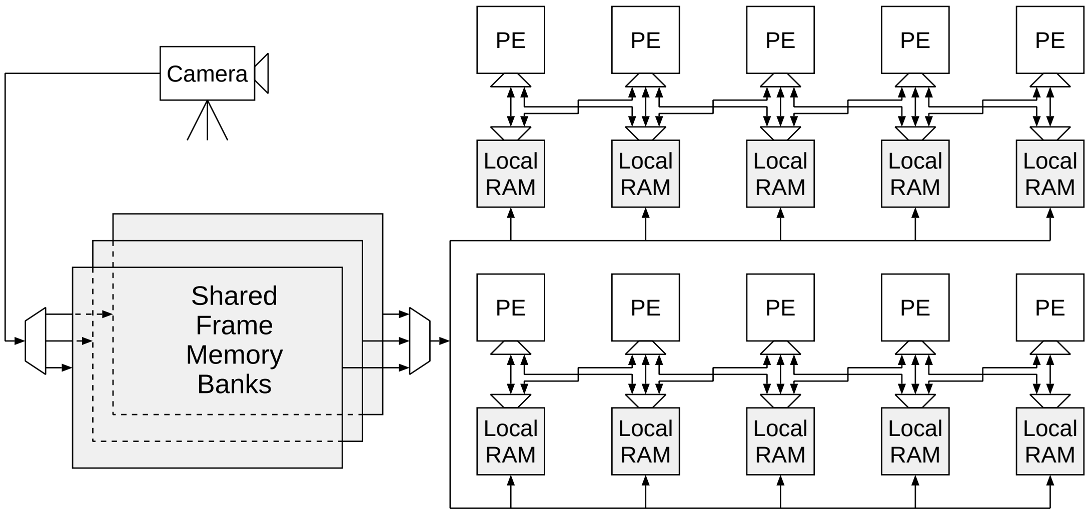

Figure 2: A parallel architecture template for block-oriented video algorithms.

Note that each of the PE's has also access to the local memories of its left and right neighbors. By using a slightly different (i.e. phase-shifted) execution and transfer schedule for the odd and even PE columns, we were able to avoid access conflicts for the local memories.

After deciding on the detailed execution order and data-distribution, we exactly knew which data where stored in what memory, and when these data where accessed. Given this information, we have then successfully (manually) applied storage order optimization techniques as described in this paper to obtain minimal sizes for each of the memories (both shared and local). This was done for each memory separately, as there is no interference between different memories for a given schedule[6].

For MIMD-type architectures with shared memories similar observations can be made, provided that the memory accesses can be synchronized and that their relative order is known at compile time.

5 A Pragmatic Approach

Our intention is to optimize the storage order for each array, such that the required size of each memory is minimal. However, as stated above, the constraints that have to be satisfied can be very hard to evaluate in practice. Therefore, an optimization strategy that takes them explicitly into account would probably not be feasible for realistic problems, so we have to use a more pragmatic approach that avoids the evaluation of these constraints, but that may lead to suboptimal results.

Without loss of generality, we will concentrate on the size reduction of only one (shared) memory. In general, multiple memories can be present, but our techniques can be applied to each memory separately, as there is no interference between the optimizations for different memories for a given data-distribution and execution order.

[6]Of course, due to symmetry reasons, it was sufficient to perform this step only for a few memories, and apply the results to the similarly accessed memories.

5.1 Observations

We can identify two components in the storage order of arrays:

- the *intra*-array storage order, which refers to the internal organization of an array in memory (e.g. row-major or column-major layout);
- the *inter*-array storage order, which refers to the relative position of different arrays in memory (e.g. the offsets, possible interleaving, ...).

This observation has stimulated us to come up with a two-phase approach. In a first phase, we try to find an optimal intra-array storage order for each array separately. This storage order then translates into a partially fixed address equation, which we will refer to as the *abstract address equation* (AAE). In the second phase, we look for an optimal inter-array storage order, resulting in a fully fixed address equation for each array. This equation will be referred to as the *real address equation* (RAE).

We will now outline a few strategies that can be used to obtain a RAE for each array, given their AAE's and corresponding OATD's (i.e. the second phase). However, in this paper we concentrate on the first phase, while a detailed discussion of the techniques and heuristics being used for tackling the second phase will be available in a future paper [8].

5.2 Inter-array storage order

In general, the exact shape of the (B)OATD's is not known, as we only have implicit descriptions. It is however possible to extract certain properties (e.g. the width or the height of an OATD) which allow us to approximate the shape of the OATD's. These approximations can then be used in several ways, depending on their accuracy, as indicated next.

In figure 3a, the OATD's of 5 different arrays are shown. The simplest way to allocate memory for these arrays is to assign a certain address range to each array in such a way that the different address ranges do not overlap. The RAE then equals the AAE, shifted by a constant offset, as illustrated in figure 3b. We will refer to this first strategy as *static allocation*. Note that this approach results in no memory savings at all. The required memory size actually equals the sum of the sizes of the arrays. This is also the approach taken by traditional compilers.

A potentially better strategy is illustrated in figure 3c. Here a certain address range is allocated for each array, but only during the time that the array is in use. This allows sharing of certain address ranges by more than one array. We will refer to this strategy as *dynamic allocation*[7]. In general, a dynamic strategy requires less memory. Also note that this strategy actually approximates the OATD's of the arrays by rectangles[8].

These first two strategies have in common that the size of the address range assigned to an array equals the size of the array, and that the intra-array storage order (which influences the shape of the OATD's) has no effect on the total memory size.

However, in general an array does not use its complete address range all the time. This has lead to the definition of an *(address reference) window* [9] as the maximum distance between two addresses being occupied by the array during its life-cycle. This address reference window is indicated for each of the arrays in figure 3a by means of a vertical arrow. If we know the size

[7]Note that this allocation strategy can be performed at *compile* time, in contrast to the traditional run-time dynamic allocation provided by certain languages such as C.

[8]In [8], we discuss some extensions though.

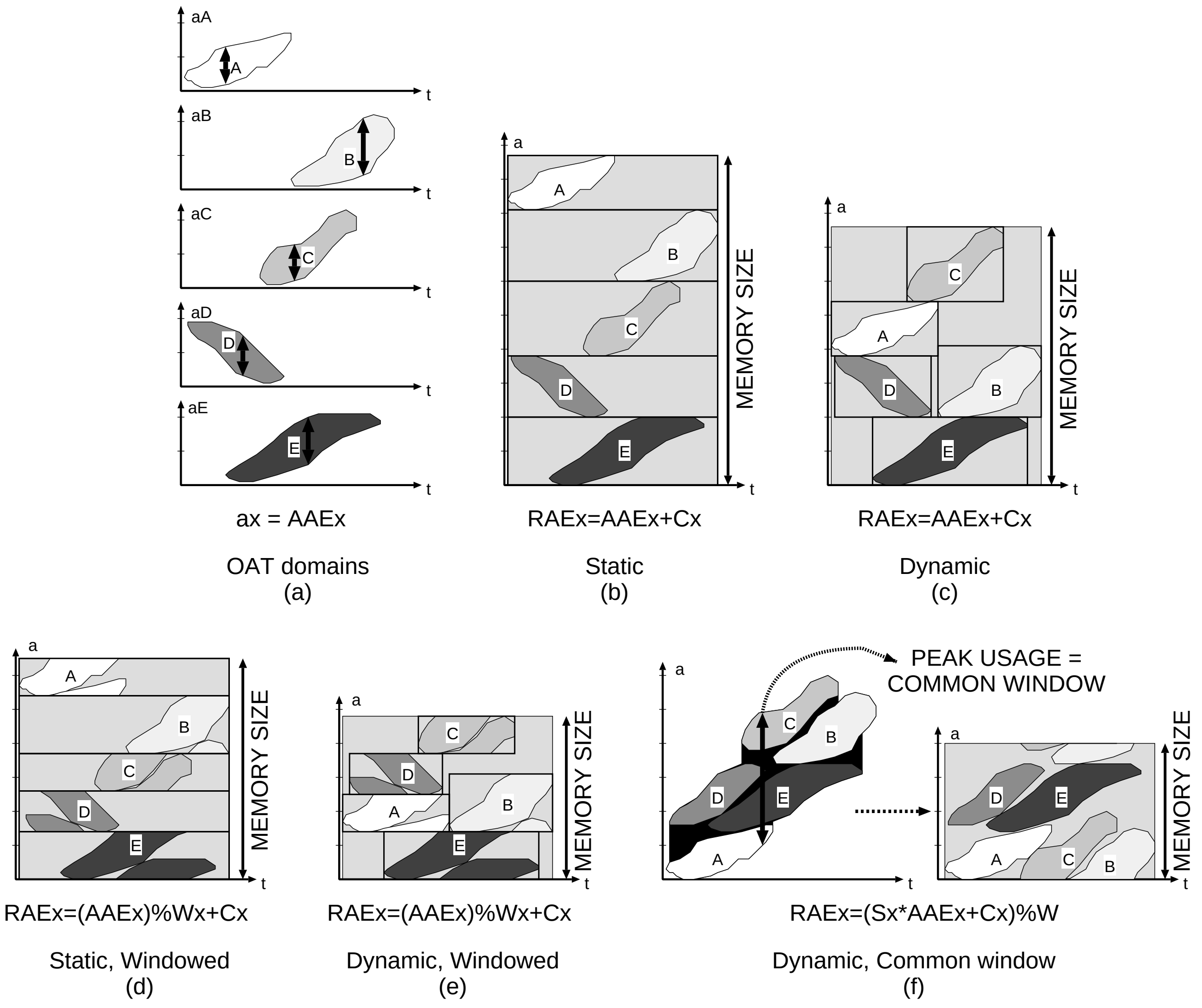

Figure 3: Different allocation strategies

of the window, we can "fold" the OATD's of the arrays by applying a modulo operation to the AAE's. After the folding, we can apply either the static or the dynamic allocation strategy, as indicated in figure 3d and figure 3e. Note that the windowed strategies in general require less memory than the non-windowed ones. They require the calculation of the address reference window though. Moreover, the internal storage order of the arrays influences the shape of the OATD's and hence the size of the address reference window. In section 6 we describe techniques for exact evaluation of the window and optimization of the intra-array storage order to obtain minimal window sizes.

A possibly even better strategy is depicted in figure 3f. In a first step, the OATD's are shifted (and possibly even vertically scaled or flipped) such that their common window is minimal. After that, the complete address range is folded by this one window. Note that for this example, the last strategy is the best one, but this is *not* true in general. The strategy with separate windows can sometimes yield better results (e.g. when the OATD's don't "fit" together very well). Moreover, the common window strategy requires the evaluation of the abovementioned very complex constraints, i.e. one has to make sure that the OATD's do not overlap, otherwise different arrays would simultaneously use the same memory locations, which is obviously illegal.

Therefore, we will concentrate on the strategy with separate windows for each array, as depicted in figure 3e, as it offers the best compromise between optimality and complexity. Our detailed placement strategy is discussed in [8], but some of the results we obtained are also presented in section 7.

5.3 Intra-array storage order

From the discussion of the inter-array storage order strategies, we can derive that we have to try to find the intra-array storage orders that result in OATD's that are as "thin" as possible, i.e. that have the smallest address reference window. The number of possible storage orders is huge however, even if we restrict ourselves to the affine ones. Moreover, checking whether a storage order is valid generally requires the evaluation of the abovementioned complex constraints and to our knowledge no practically feasible strategy exist for choosing the best order.

Therefore, we will restrict the number of possibilities drastically. First of all, we will require that each element of an array is mapped onto an abstract address that is unique w.r.t. the address of the other elements of the same array. In that way, we can avoid checking for intra-array memory occupation conflicts[9]. Another requirement we impose is that the storage order should be dense, i.e. the set of abstract addresses occupied by a rectangular array should be a closed interval. A row-major order as in C for instance, satisfies this requirement. However, for multi-dimensional arrays, we will consider *all* possible orders of the dimensions, and also both directions (i.e. positive and negative) for each dimension. Consequently each AAE will have the following format and properties:

$$\begin{array}{rcl} AAE_x & = & N_{1x}(\mp B_{n_1x} \pm a_{n_1x}) + N_{2x}(\mp B_{n_2x} \pm a_{n_2x}) + ... + N_{Dx}(\mp B_{n_Dx} \pm a_{n_Dx}) \\ N_{1x} & = & \mathtt{max}(N_{2x}(\mp B_{n_2x} \pm a_{n_2x}) + ... + N_{Dx}(\mp B_{n_Dx} \pm a_{n_Dx})) = N_{2x}N_{3x}...N_{Dx} \geq 1 \\ N_{2x} & = & \mathtt{max}(N_{3x}(\mp B_{n_3x} \pm a_{n_3x}) + ... + N_{Dx}(\mp B_{n_Dx} \pm a_{n_Dx})) = N_{3x}...N_{Dx} \geq 1 \\ & ... & \\ N_{Dx} & = & 1 \end{array} \quad (1)$$

Here, a_{n_ix} and B_{n_ix} represent the index and the upper or lower bound of dimension n_i of array x respectively. The N_{ix} coefficients are constants, obeying certain criteria to obtain dense storage orders. The signs depend on the chosen dimension directions and D equals the total number of dimensions of the array. An example of the possible orders we consider for a 2x3 2-D array is given in figure 4.

In general, for a N-dimensional array, we consider $2^N N!$ possibilities. For a 6-D array[10] for instance, there are no less than 46080 possibilities! It must be clear that even though this is only a very limited subset of all possible storage orders, evaluating this set will be infeasible for arrays with many dimensions. So we have to come up with a more intelligent search strategy. But first we will describe how we can evaluate the size of a window for a given storage order.

[9] Note that we only require this for the *abstract* addresses. Later on, due to the possible folding, several elements may use the same real addresses, provided that they have non-overlapping life-times. Moreover, one can always split arrays at the specification level to provide more freedom (or merge/interleave them to limit the freedom).

[10] A 6-D array is not unusual for single-assignment code.

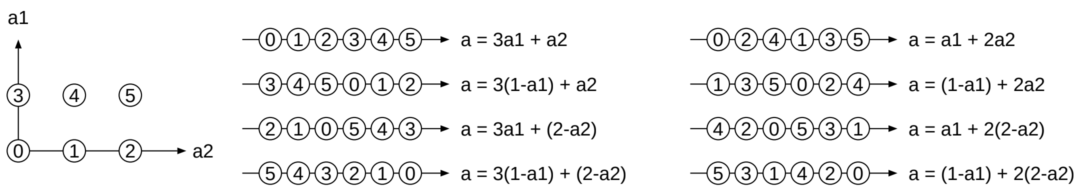

Figure 4: The possible dimension orders for a 2x3 2-D array.

6 Address Reference Window Optimization

6.1 Address Reference Window Calculation

Conceptually, the evaluation of the address reference window for a given intra-array storage order is relatively simple. We have to find the maximal distance between two addresses being occupied by the array at the same time. The BOATD descriptions allow us to calculate this distance by solving a finite number of ILP problems. More in particular, for each pair of BOATD's $(BOATD_1, BOATD_2)$, we have to calculate $max(|a_1 - a_2|)$ for $(a_1, t) \in BOATD_1$ and $(a_2, t) \in BOATD_2$. The overall maximum + 1 is then equal to the size of the address reference window. This strategy is depicted for an array with 3 dependences (and consequently 3 BOATD's) in figure 5. Note that the calculation of one maximum requires the evaluation of 2 ILP problems (due to the absolute value)[11]. Moreover, if an array has D dependences, then we have to calculate *at most* $D(D+1)/2$ maxima, or evaluate *at most* $D(D+1)$ ILP problems[12], which is acceptable in practical cases (see section 7).

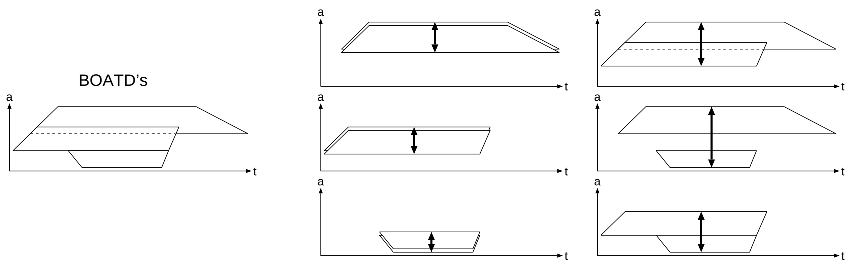

Figure 5: The address reference window calculation.

6.2 Choice of Optimal Intra-Array Storage Order

Now that we know how to calculate the window size of an array, we can try to find the optimal order and direction of the different dimensions. A first brute-force strategy can be based on the evaluation of the window for each possible order. For arrays with only a few dimensions, this strategy is generally feasible, but when the array has more dimensions (e.g. 6 or more), the number of possibilities to evaluate can become prohibitively large. Remember that for an

[11]The number of variables and constraints in the ILP problems depends on the number of dimensions of the array and the depth of the loop nests around the array accesses. In practice, there are usually at most a few dozen variables and constraints.

[12]At most, because sometimes we can avoid a (large) number of evaluations (e.g. if there's no solution for a1-a2, then we don't have to look for a solution for a2-a1).

array with N dimensions and D dependences, at most $2^N N! D(D+1)$ ILP problems have to be solved.

A better strategy can be based on the following observation: given that we have fixed the order and direction of some of the dimensions, we can already calculate an upper and a lower bound on the window, no matter what the order and direction of the remaining dimensions is. This is possible due to the special properties of the AAE presented in equation 1 and can be understood as follows.

Supposing that we have already chosen the first dimension and its direction, we can calculate a window for the corresponding *partial* AAE: $W_{1x} = window(N_{1x}(\mp B_{n_1 x} \pm a_{n_1 x}))$. One can easily prove that $W_{1x} - N_{1x} + 1 \leq window(AAE_x) \leq W_{1x}$. Similarly, if we have fixed the first two dimensions, then $W_{2x} - N_{2x} + 1 \leq window(AAE_x) \leq W_{2x}$. Due to the decreasing values of the N_{ix} constants, the distance between the upper and lower bounds will gradually decrease as we fix more dimensions. Eventually, when all dimensions have been fixed, the upper and lower bound will coincide. This property is ideally suited for being used in a branch-and-bound (B&B) strategy, because large parts of the search tree can usually be pruned. Of course the worst-case complexity ($N! \sum_{i=1}^{N} \frac{2^i}{(N-i)!}$ alternatives) of a full B&B strategy is worse than that of a full search ($2^N N!$ alternatives), but in practice it is highly unlikely that the complete B&B tree will have to be searched as there are usually some dominant dimensions. In our experiments (see section 7), the number of evaluated nodes in the search tree was always reasonable.

We have also evaluated a more greedy, but heuristic B&B search strategy, in which we follow only the most promising branch at each node in the search tree. In that case, only $N(N+1)$ window evaluations are necessary, and even that has always lead to very good and most of the time even optimal results (within the boundaries of our search space).

In our implementation for each of the considered search strategies, we have introduced several optimizations (usually based on common data-flow analysis techniques) that enable us to reduce the number of ILP problems to be solved and/or their complexity drastically in practice[13], but due to space limitations, we cannot discuss them here.

7 Experimental Results

Our size reduction strategy is intended to be applied to each memory separately after the traditional data-distribution and transfer scheduling steps, i.e. when we know the relative order of the accesses to each (shared) memory. The actual *nature* of the execution order (i.e. parallel or sequential) is not relevant for our techniques, as we are only interested in the relative memory access order. We therefore assumed in each of our experiments that all of the arrays had to be stored in one (shared) memory, and that the applications had to be executed sequentially, but the techniques would work equally well in case of a parallel execution order.

In table 1 we present the most relevant properties of a the multimedia applications and application kernels that we used as test vehicles, namely an updating singular value decomposition algorithm, a 2-D wavelet compression algorithm, an edge detection algorithm, a 3-D volume rendering algorithm, a voice coder algorithm, and a public domain GSM autocorrelation algorithm. The table contains a.o. the required array memory sizes for the original multiple-assignment versions and the single-assignment versions of the algorithms (with the same assumptions: one

[13] E.g. for the B&B strategies, if the bound is exceeded during the evaluation of the window for a certain node, we simply abort the evaluation.

memory and a sequential execution order). The last column also indicates the maximal number of scalars that is simultaneously alive during the execution of the algorithms (obtained by symbolic simulation). This column represents a lower bound on the required memory sizes, i.e. the size that would be required if all the arrays would be split in scalars, a solution that is obviously unacceptable for realistic multimedia applications, as the control and address generation overhead would be prohibitively large[14].

Application	# Arrays	Max. #Depend. per Array	Max. #Dimens. per Array	Multiple Assignment [words]	Single Assignment [words]	Scalar Minimum [words]
Updating SVD	6	27	4	6013	6038	211
2D wavelet	11	18	5	1186	8704	514
Edge detection	18	17	4	724	5647	116
3D volume rendering	22	11	8	26581	216M	infeas.
Reduced 3D volume rend.	22	11	8	166	6976	134
Voice coder	201	41	6	2963	314K	905
GSM autocorrelation	17	35	3	532	1279	209

Table 1: Relevant application properties

We then let our prototype tool decide on an optimal storage order for each array (using different strategies). The resulting memory sizes (in terms of the number of words) and the optimization run-times of our prototype tool are indicated in table 2. We compared five static windowed allocation strategies and a dynamic approach. In the first two static strategies, we assumed a fixed internal storage order (i.e. row-major or column-major) that we combined with an address reference window. In the remaining three static strategies, we also explored other intra-array storage orders, using different search techniques. Finally, the last column contains the results for a dynamic strategy which is based on the full branch-and-bound static one. The additional techniques and heuristics used in our dynamic allocation strategy are described in a future paper [8].

From this table, we can see that the savings in memory size for the windowed strategies can be considerable, especially for the dynamic strategy, while the run-times are certainly acceptable in an embedded application design context. It also shows that an arbitrary choice of the storage order (i.e. row-major or column-major) can easily lead to suboptimal results. The row-major storage order performs quite well for most of the experiments, but this can be explained by the fact these examples have been written manually, and humans seem to tend to access arrays in a row-major manner. In our ATOMIUM memory and power optimization context [7] however, the memory size reduction step follows several other steps that make

[14]For larger examples, such as the 3D volume rendering, even symbolic simulation is infeasible.

Applic.	Static Windowed Allocation (only intra-array optimization)					Dynamic
	Row-major [words] [seconds]	Col.-major [words] [seconds]	Full Search [words] [seconds]	Full B&B [words] [seconds]	Heur. B&B [words] [seconds]	Wind. All. [words] [seconds]
Updating SVD	3067 59	484 59	314 84	314 74	314 73	312 74
2D wavelet	3500 47	8348 49	3038(1024) 66	3038(1024) 57	3038(1024) 56	2846(832) 72
Edge detection	580 45	1021 45	576 46	576 45	576 45	189 47
3D vol. render.	26576 263	216M 267	26576 355	26576 292	26576 282	25603 295
Reduc. 3D vol. rend.	166 199	4756 197	166 227	166 213	168 208	147 216
Voice coder	2417 942	38537 928	2403 1032	2403 1004	2403 997	1130 1624
GSM autocorr.	667 58	1096 56	529 61	529 59	529 59	248 78

Table 2: Experimental results: memory sizes and optimization run-times.

extensive use of loop transformations. After these loop transformations, the array access order can be changed drastically and consequently the row-major order is in general no longer likely to be near-optimal. The updating SVD algorithm is such an example that has been subject to loop transformations before our optimizations.

In contrast to what might be expected, the full search and B&B techniques have comparable run-times, even for applications with arrays with a large number of dimensions. The reason why the full search strategy run-times do not explode even in these cases, is that we first perform an analysis which can detect dimensions that can be completely "collapsed", i.e. dimensions that would have no effect on the window size for the optimal storage order, as they would always "end up on the outside". Usually only a few dimensions remain, such that an explosion of the number of combinations does not occur. Nevertheless, the B&B techniques tend to be faster and they are probably the best choice. The full B&B version is probably preferable as it guarantees the optimal solution (within the search space), in contrast to the heuristic B&B.

The optimization run-times also turn out to be relatively independent of the size parameters of the applications (e.g. the sizes of the loop bounds), as indicated by the results obtained for the 3D volume rendering application. The run-times for the original specification are comparable to those obtained for a version with (heavily) reduced size parameters, even though the number of array elements differs orders of magnitude.

Also note that in most cases we were able to effectively remove the single-assignment overhead, and that for these cases we usually obtain results that are substantially better than what would be obtained by a standard compiler (i.e. the multiple assignment column in table 1). Only for one example, namely the 2D wavelet algorithm, we could not remove the overhead. Detailed inspection of the example revealed that for two arrays one of the dimensions could not be "collapsed" with our techniques due to a special access pattern. By extending our techniques with a projection approach as described in [14], or by applying a loop transformation in advance, this overhead can be removed too, and this would result in a memory size of 1024 words for the best static approaches and 832 words for the dynamic approach (indicated between brackets).

Apart from the single-assignment overhead removal, the additional size reductions are definitely worthwhile. For the updating SVD example for instance, no "trivial" collapsing could be done. The tool "s2p", that implements the strategy described in [9], and to our knowledge the only other tool that tries to perform our type of optimizations[15], could not reduce the memory usage to less than 3285 words[16], while we were able to reduce it to 312 words.

Finally, the table also shows that a dynamic approach can result in a considerable gain compared to a static approach that is also taken by other researchers [14].

8 Conclusion

In this paper we have presented a two-phase strategy that is able to reduce the memory size requirements for arrays in static control single assignment programs with a static schedule. We have concentrated on the first phase which deals with intra-array storage reduction. The effectiveness and feasibility this strategy has been demonstrated on several relevant multimedia algorithms and can be very valuable in an embedded (parallel) multimedia application context.

References

[1] F. Catthoor, W. Geurts, and H. De Man. Loop transformation methodology for fixed-rate video, image and telecom processing applications. In *Proc. Int. Conf. on Application Specific Array Processors*, pages 427–438, San Francisco, CA, Aug. 1994.

[2] R. Gonzalez and M. Horowitz. Energy dissipation in general-purpose microprocessors. *IEEE J. Solid-state Circ.*, SC-31(9):1277–1283, Sep. 1996.

[3] T. H. Meng, B. Gordon, E. Tsern, and A. Hung. Portable video-on-demand in wireless communication. *Proc. of the IEEE, special issue on Low power electronics*, 83(4):659–680, April 1995.

[4] L. Nachtergaele, F. Catthoor, B. Kapoor, S. Janssens, and D. Moolenaar. Low power storage exploration for H.263 video decoder system. In W. Burleson, K. Konstantinides, and T. Meng, editors, *VLSI Signal processing IX*, pages 115–124. IEEE press, October 1996.

[5] S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man. Power exploration for data dominated video applications. In *Proc. Int. Symposium on Low Power Electronics and Design*, pages 359–364, Monterey, USA, August 1996.

[6] E. De Greef, F. Catthoor, and H. De Man. Reducing storage size for static control programs mapped onto parallel architectures. In *Dagstuhl Seminar on Loop Parallelization*, Dagstuhl, Germany, Apr. 1996.

[15] Of course, several scalar memory/register reuse approaches exist, but it must be clear that these approaches are infeasible for data-intensive multimedia applications, where the number of scalars is huge.

[16] The tool actually calculates an upper-bound for the windowed row-major storage order.

[7] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man. Global communication and memory optimizing transformations for low power signal processing systems. In *Proc. IEEE workshop on VLSI signal processing*, La Jolla, CA, Oct. 1994. Also in *VLSI Signal Processing VII*, J. Rabaey, P. Chau, J. Eldon (eds.), IEEE Press, New York, pp.178-187, 1994.

[8] E. De Greef, F. Catthoor, and H. De Man. Array placement for storage size reduction in embedded multimedia systems. Submitted to the 11th International Conference on Application-specific Systems, Architectures and Processors, July 1997.

[9] J. Vanhoof, I. Bolsens, and H. De Man. Compiling multi-dimensional data streams into distributed DSP ASIC memory. In *Proc. IEEE Int. Conf. on Computer Aided Design*, 1991.

[10] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man. In-place memory management of algebraic algorithms on application specific ic's. In E. F. Deprettere and A.-J. van der Veen, editors, *Algorithms and Parallel VLSI Architectures, Volume B: Proceedings*, pages 353–362. Elsevier Science Publishers, 1991.

[11] F. Balasa, F. Catthoor, and H. De Man. Exact evaluation of memory size for multi-dimensional signal processing systems. In *Proc. ICCAD'93*, pages 669–672, Santa Clara, CA, Nov. 1993.

[12] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[13] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proc. of the SIGPLAN'95 Conf. on Programming Language Design and Implementation*, 1995.

[14] S. Rajopadhye and D. Wilde. Memory reuse analysis in the polyhedral model. In *Dagstuhl Seminar on Loop Parallelization*, Dagstuhl, Germany, Apr. 1996.

[15] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Proc. Euro-Par'96*, Lyon, France, Aug. 1996.

[16] E. De Greef, F. Catthoor, and H. De Man. Mapping real-time motion estimation type algorithms to memory efficient, programmable multi-processor architectures. *Microprocessing and Microprogramming*, 41:409–423, 1995.