Loop Transformation Methodologies for Array-Oriented Memory Management[*]

F. Balasa[*] P.G. Kjeldsberg[†] M. Palkovic[‡] A. Vandecappelle[‡] F. Catthoor[‡§]

[*] Dept. of Computer Science, University of Illinois at Chicago, Chicago, U.S.A. fbalasa@cs.uic.edu
[†] Norwegian University of Science and Technology, Trondheim, Norway pgk@iet.ntnu.no
[‡] IMEC vzw, Leuven, Belgium {vdcappel,palkovic,catthoor}@imec.be
[§] also professor at Katholieke Universiteit Leuven, Belgium

Abstract – The storage requirements in data-dominant signal processing systems, whose behavior is described by array-based, loop-organized algorithmic specifications, have an important impact on the overall energy consumption, data access latency, and chip area. Applying different loop transformations on the specification code can significantly enhance the memory management of such VLSI systems, improving all the major parameters of the design space – power, area, and performance. This paper gives a global view on existing and recently proposed memory size evaluation approaches for procedural and non-procedural specifications. Moreover, it discusses typical memory management trade-offs taken into account during the exploration of system specifications by loop transformations, that can exploit these early size evaluations.

1 Introduction and Motivation

In many signal processing systems, particularly in the multimedia and telecom domains, data transfer and storage have a significant impact on both the system performance and the major cost parameters – power consumption and chip area. During the system development process, the designer must often focus first on the exploration of the memory subsystem in order to achieve a cost optimized end product [9, 10].

The behavior of these targeted VLSI systems, synthesized to execute mainly data-dominant applications, is described in a high-level programming language, where the code is typically organized in sequences of loop nests having as boundaries (usually affine) functions of loop iterators, conditional instructions where the arguments may be data-dependent and/or data-independent (relational and/or logic expressions of affine functions of loop iterators). In our target domain, the data structures are multi-dimensional arrays whose indexes in the code are affine functions of loop iterators. The class of specifications with these characteristics are often called *affine* specifications [9].

Global loop transformations are important system-level design techniques, used to enhance the locality of data and the regularity of data accesses in affine specifications. Reducing the lifetime of the array elements increases the possibility of memory sharing, since data with non-overlapping lifetimes can be mapped to the same physical location. This leads to the overall reduction of the data storage requirements and, hence, of the chip area. Also, due to the large amounts of data in our targeted applications, both on-chip and off-chip memory modules are usually needed. Improving data locality by global loop reorganization enhances the data reuse [34, 20], globally reducing the off-chip memory accesses, which is critical for system performance and energy consumption [10].

After an overview of the past works addressing the memory estimation/computation problem (Section 2), this paper presents (in Section 3) two recent models for the evaluation of storage requirements. The first approach does an accurate estimation exploring the possibilities of reordering the loop execution aiming to optimize data locality. The second technique performs an exact computation after the loop execution has already been fixed. Based on these high-level requirements, some memory management trade-offs that must be taken into account during the exploration of the design space are discussed (in Section 4), as well. Section 5 states the main conclusions of this work, performed in several research centers.

2 The Memory Size Computation Problem: A Brief Overview

For several decades, researchers have worked on different approaches for estimating or computing the memory size required to execute a given application. Most of the work is focused on scalars. The number of scalars, also called signals or variables, is then limited, and if arrays are treated, they are flattened and each array element is considered a separate scalar. Using scheduling techniques like the left-edge algorithm, ASAP-ALAP, force directed scheduling, partitioning techniques like clique partitioning, and integer linear programming based techniques, the number of memory locations required is found [18]. Common to all scalar based techniques is that they break down when used for large multi-dimensional arrays. The problem is NP-hard and its complexity grows exponentially with the number of scalars. When the multi-dimensional arrays present in the applications of our target domain are flattened, they result in many thousands or even millions of scalars.

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

To overcome the shortcomings of the scalar-based techniques, several research teams have tried to split the arrays into suitable units before or as a part of the estimation. Typically, each instance of array element accessing in the code is treated separately. Due to the loop structure of the code, large parts of an array can be produced or consumed by the same code instance. This reduces the number of elements the estimator must handle compared to the scalar approach. We will now present different published contributions using this approach, starting with techniques that assume a procedural execution of the application code.

In [38], a production time axis is created for each array. This models the relative production and consumption time, or date, of the individual array accesses. The difference between these two dates equals the number of array elements produced between them. The maximum difference found for any two depending instances gives the storage requirement for this array. The total storage requirement is the sum of the requirements for each array. An Integer Linear Programming approach is used to find the date differences. Since each array is treated separately, only in-place mapping internally to an array (intra-array in-place) is consider, not the possibility of mapping arrays in-place of each other (inter-array in-place). With in-place mapping we mean the optimization technique where data with non-overlapping lifetimes can be mapped to the same physical memory locations [37].

Another approach is taken in [19]. The data-dependency relations between the array references in the code are used to find the number of array elements produced or consumed by each assignment. The storage requirement at the end of a loop equals the storage requirement at the beginning of the loop, plus the number of elements produced within the loop, minus the number of elements consumed within the loop. The upper bound for the occupied memory size within a loop is computed by producing as many array elements as possible before any elements are consumed. The lower bound is found with the opposite reasoning. From this, a memory trace of bounding rectangles as a function of time is found. The total storage requirement equals the peak bounding rectangle. If the difference between the upper and lower bounds for this critical rectangle is too large, better estimates can be achieved by splitting the corresponding loop into two loops and rerunning the estimation. In the worst-case situation, a full loop-unrolling is necessary to achieve a satisfactory estimate.

Reference [42] describes a methodology for so-called exact memory size estimation for array computation. It is based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. In this context, a polytope is the intersection of a finite set of half-spaces and may be specified as the set of solutions to a system of linear inequalities. It is shown that it is only necessary to find the number of live variables for one statement in each innermost loop nest to get the minimum memory size estimate. The live variable analysis is performed for each iteration of the loops however, which makes it computationally hard for large multi-dimensional loop nests.

In [30], a reference window is used for each array in a perfectly nested loop. At any point during execution, the window contains array elements that have already been referenced and will also be referenced in the future. These elements are hence stored in local memory. The maximal window size found gives the memory requirement for the array. If multiple arrays exist, the maximum reference window size equals the sum of the windows for individual arrays. Inter-array in-place is consequently not considered.

All the techniques above estimate the memory size assuming a single memory. [20] performs hierarchical memory size estimation, taking data reuse and memory hierarchy allocation into account. In-place mapping is not incorporated in the current version, but is indicated as part of future work.

In contrast to the array based methods described so far in this section, the storage requirement estimation technique presented in [4] assumes a non-procedural execution of the application code. It traverses a dependency graph based on an extended data dependency analysis resulting in a number of non-overlapping array sections (so called basic sets) and the dependencies between them. The basic set sizes and the sizes of the dependencies are found using an efficient lattice point counting technique [5]. The maximal combined size of simultaneously alive basic sets found through a greedy graph traversal gives the storage requirement.

The techniques described above are mainly used at an early design stage to estimate the memory size required for the final implementation. In addition to this, much work has been focused on performing the final in-place optimization and signal-to-memory mapping. This is not the main focus of this paper, though. See [13] for a good overview of the work in this field.

In the next section we will present some more details for two alternative techniques that can be used to find the storage requirement of an application. We will then also describe the differences between these two techniques and the techniques presented in this section.

3 Non-Scalar Models for Memory Size Evaluation

3.1 Estimation Approach for Non-Procedural Specifications

All but the last estimation approach described in the previous section assume a fully fixed ordering. This makes them hard to use during early system level design steps such as the global data flow transformations and global loop transformations [10]. The number of alternative solutions is huge, each with a different execution ordering. It is impossible to evaluate each of these using its fixed ordering. The designer needs size estimates that work with only a partially fixed execution ordering. On the other hand, the last approach of the previous section does not take available execution ordering into account at all. It therefore produces the same estimate both before and after a transformation that fixes parts of the ordering. It can consequently not be used for guidance.

For our target classes of data dominant applications the high level description is typically characterized by large multi-dimensional loop nests and arrays with mostly manifest index expressions and loop bounds. *Example 1* shows the code of a simple loop nest. Two statements, *S1* and *S2*, produce elements of two arrays, *A* and *B*. Elements from array *A* are consumed when elements of array *B* are produced. This gives rise to a flow type data dependency between *S1* and *S2* [2]. In this example the array index functions are simple, but the techniques presented in this section work for any affine and manifest index functions. Section 4 discusses preprocessing techniques that can be employed if this is not the case. We also require the dependencies in the code to be linear. If this is not the case, different linearization techniques can be used [25].

Example 1:
```
 for (x=0; x<=5; x++)
   for (y=0; y<=5; y++)
      for (z=0; z<=2; z++) {
S1:   A[x][y][z] = f1(input);
S2:   if (x>=1 && y>=2) B[x][y][z]=f2(A[x-1][y-2][z] ;
      }
```

Loop interchange is a transformation with very large impact on lifetimes of data elements within loop nests. With the worst case ordering of the dimensions in *Example 1*, y outermost, x second outermost, and z innermost (y, x, z), the storage requirement for the dependency between *S1* and *S2* is 33 memory locations. For the best case ordering, (z, x, y), the requirement is 6. For this simple example, the absolute numbers are small. The ratio between them is large, however, and this holds also for large real life examples.

We will now give an overview of the four steps of our estimation methodology. It takes available partially fixed ordering into account to find upper and lower bounds (UB and LB) on the run time storage requirement of an application. The span between the bounds reflects the still unfixed part of the ordering.

Step 1 *Collect DPs and data dependencies from application code.*

Our estimation methodology uses a geometrical model to describe data, operations and dependencies. This first step is performed by the Atomium tool [44], based on the *Iteration Domains* (ID) of the statements and the index functions in the application code. Fig. 1 shows graphically the *iteration space* [2] of the loop nest in *Example 1* with the IDs of the two statements. At each *iteration node* within ID_{S1} and ID_{S2}, statement *S1* and *S2* are executed, respectively. It is, however, only a subset of the elements produced by *S1* that are read by *S2*. A *Dependency Part* (DP) is therefore defined as shown in Fig. 1 containing the iteration nodes at which elements are produced that are read by the depending statement. A *Dependency Vector* (DV) is drawn from an iteration node in the DP producing an array element to the iteration node consuming the element. This DV spans a *Dependency Vector Polytope* (DVP) and its dimensions are defined as *Spanning Dimensions* (SD). The remaining dimensions are denoted *Non-spanning Dimensions* (ND). In Fig. 1, *x* and *y* are SDs while *z* is ND. More details about these concepts can be found in [24].

Step 2 *Position DPs in a common iteration space.*

The IDs are placed in a common iteration space to enable a global estimation scope even for applications with loops that are not perfectly nested [25]. A best case and worst case common iteration space may be necessary to find the memory size LB and UB for the complete application.

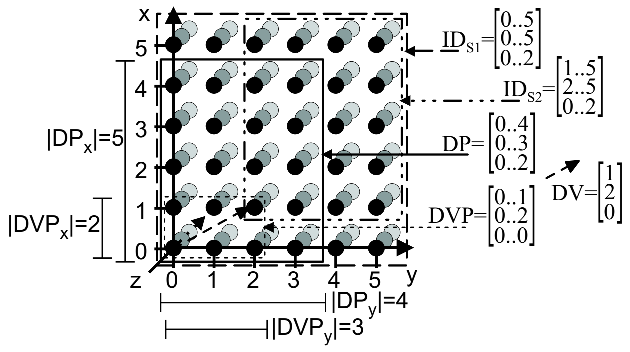

x
5
4
3
2
1
0
z
0 1 2 3 4 5
y
$|DP_x|=5$
$|DVP_x|=2$
$|DP_y|=4$
$|DVP_y|=3$
$ID_{S1}=\begin{bmatrix}0..5\\0..5\\0..2\end{bmatrix}$
$ID_{S2}=\begin{bmatrix}1..5\\2..5\\0..2\end{bmatrix}$
$DP=\begin{bmatrix}0..4\\0..3\\0..2\end{bmatrix}$
$DVP=\begin{bmatrix}0..1\\0..2\\0..0\end{bmatrix}$
$DV=\begin{bmatrix}1\\2\\0\end{bmatrix}$

Figure 1: Iteration space and IDs from *Example 1*.

Step 3 *Estimate UB and LB on the size of individual dependencies based on the partially fixed execution ordering.*

The size of a dependency between two IDs equals the number of iteration nodes visited in the DP before the first depending iteration node is visited. Since one array element is produced at each iteration node in the DP, this size equals the number of array elements produced before the first depending array element is produced that potentially can be mapped in-place of the first array element.

We will now present a number of guiding principles for the ordering of loops in a loop nest. The size of a dependency is minimized if the execution ordering is fixed so that its NDs and SDs are placed at the outermost and innermost nest levels respectively. The order of the NDs is of no consequence as long as they are all fixed outermost. If one or more SDs are ordered in between the NDs, it is however better to place the shortest NDs inside the SDs. The length of ND i_k is determined by the length of the DP in this dimension $|DP_k|$. The ordering of SDs is important even if all of them are fixed innermost. The SD with the largest Length Ratio (LR) should be ordered innermost. The LR is defined as follows:

$$LR_k = \frac{|DVP_k| - 1}{|DP_k| - 1}$$

For the special case when $|DP_k| = 1$, care must be taken to avoid division by zero. An $|LR_k| = \infty$ can still be assumed, since such dimensions should be ordered innermost.

Returning to the dependency between *S1* and *S2* in *Example 1*, the LRs for the SDs are easily calculated. For each dimension we use $|DP|$ and $|DVP|$ as illustrated in Fig. 1. This gives $LR_x = 1/4$ and $LR_y = 2/3$. According to the guiding principles, the *y* dimension should consequently be fixed innermost with *x* second innermost, and ND *z* outermost. If the guiding principles are applied in their opposite order, we get the worst case ordering. Using these guiding principles for best case and worst case ordering, our estimation methodology finds the orderings that will result in the LB and UB storage requirement of individual dependencies. Any ordering already fixed will be taken into account.

Starting outermost, the contribution of each nest level to the total dependency size is calculated. If a dimension is already fixed at a given nest level, this dimension is used to calculate the contribution. If not, the dimensions according to the best case and worst case orderings are used for LB and UB calculations, respectively. When calculating the contribution of a given nest level, we multiply Q_j of the dimension fixed here with the P_k of all dimensions fixed inside it. Here $Q_j = |DVP_j| - 1$ and $P_k = |DP_k|$. The total dependency size is the sum of contributions of all nest levels. Note that $Q_j = 0$ for all NDs, so they do not contribute to the overall dependency size except with their P_k if they are fixed inside an SD. Tab. 1 demonstrates the stepwise calculation of dependency sizes with an unfixed and a partially fixed execution ordering. The different values for Q_j and P_k can be found from Fig. 1. Note that as the ordering is gradually fixed, the bounds converge. With a fully fixed ordering they are equal. In addition to the LB and UB, the best case ordering found is reported to the user. The complete algorithm is described in detail in [24].

	Completely unfixed	x fixed outermost
BC ordering	(z,x,y)	(x,z,y)
WC ordering	(y,x,z)	(x,y,z)
1st nest level	LB_t=0 UB_t=$Q_y \cdot P_x \cdot P_z$=30	LB_t=UB_t= $Q_x \cdot P_y \cdot P_z$=12
2nd nest level	LB_t=LB_t+$Q_x \cdot P_y$=4 UB_t=UB_t+$Q_x \cdot P_z$=33	LB_t=LB_t=12 UB_t=UB_t+$Q_y \cdot P_z$=18
3rd nest level	LB=LB_t+Q_y=6 UB=UB_t=33	LB=LB_t+Q_y=14 UB=UB_t=18

Table 1: Storage requirement for the code in *Example 1* with unfixed and partially fixed execution ordering.

Step 4 *Find simultaneously alive dependencies and their maximal combined size ⇒ Bounds on total storage requirement.*

After having found the upper and lower bounds on the size of each dependency in the common iteration space, it is necessary to determine if two or more of them are alive simultaneously. The maximal combined size of simultaneously alive dependencies over the lifetime of an application, gives the total storage requirement of the application. Two dependencies can potentially be alive simultaneously if their DPs overlap in one or more dimensions in the common iteration space. Depending on whether the overlap occurs only for NDs, for a subset of the dimensions including at least one SD, or for all dimensions, they will alternate in being alive, be alive simultaneously for certain execution orderings, or be alive simultaneously regardless of the chosen execution ordering. Similar reasoning can be made for groups of multiple dependencies. The estimation methodology uses a two-step procedure. First, groups of potentially simultaneously alive dependencies are detected, followed by an inspection to reveal those actually simultaneously alive for a given partially fixed execution ordering. Details regarding this part of the methodology are presented in [23].

Section 4 shows how this estimation methodology can be a part of a solution space exploration during the global loop transformation design step. Furthermore, [24, 25] gives exploration examples for real life application demonstrators.

3.2 Computation Approach for Procedural Specifications

This section presents a non-scalar method for computing *exactly* the memory size in signal processing algorithms where the code is *procedural*, that is, where the loop structure and sequence of instructions induce the (fixed) execution ordering. This assumption is based on the fact that the design entry in present industrial design usually includes a full fixation of the execution ordering. Even if this is not the case, the designer can still explore different algorithmic specifications functionally equivalent.

The exact computation of storage requirements may be necessary in embedded system applications, where the constraints on the data memory space are typically tighter. It may be useful as well in assessing the impact of different code (and, in particular, loop) transformations on the data storage. The exact computation approach allows to address the problem of signal-to-memory mapping by providing exact determinations of window sizes for multi-dimensional signals and their indexes. Finally, it allows a more precise data reuse analysis [34, 20] and, therefore, a better steering of the (hierarchical) memory allocation [9, 10].

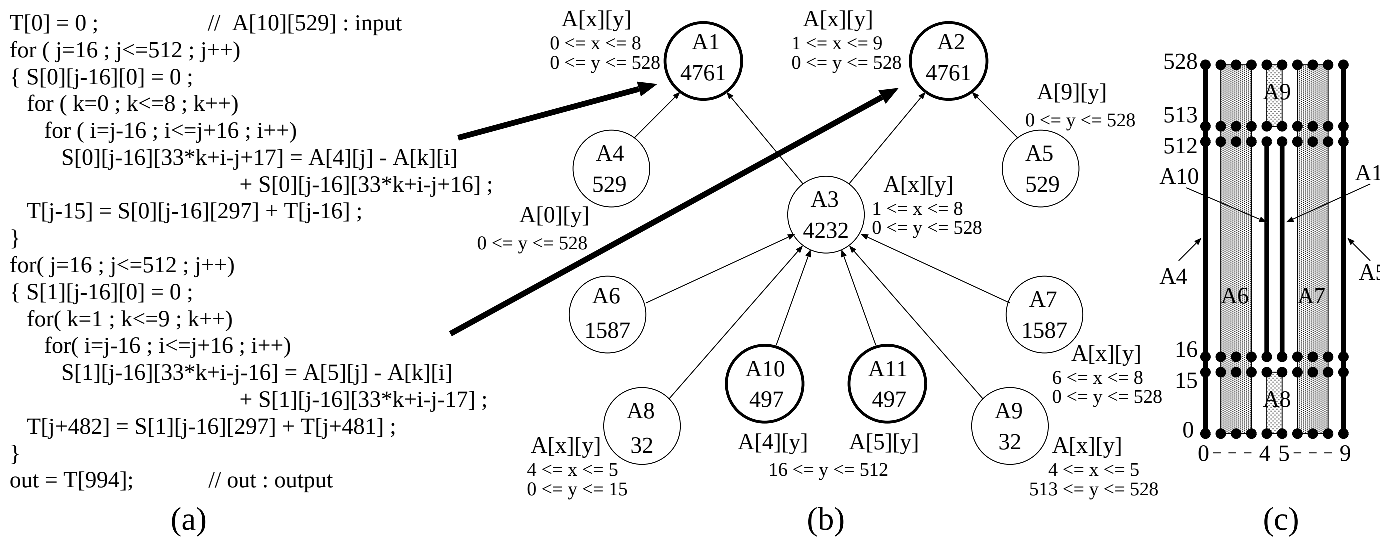
An array reference can be typically represented as the image of an affine vector function $\mathbf{i} \longmapsto \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$ over a **Z**-polytope[1] (its iterator space) $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$, therefore, a *lattice* [31] which is *linearly bounded* [33]. The main steps of the memory size computation algorithm [43] will be briefly presented below.

Step 1 *Extract the array references from the given algorithmic specification and decompose the array references of every indexed signal into* disjoint *linearly bounded lattices (LBLs).*

Figure 2(b) shows the result of this decomposition for the 2-dimensional signal A in the illustrative example from Fig. 2(a). The graph displays the inclusion relations ($arcs$) between the LBLs of A ($nodes$). The 4 "bold" nodes are the 4 array references of signal A in the code. The nodes are also labeled with the size of the corresponding LBL – that is, the number of lattice points (i.e., points having integer coordinates) in those sets. The *inclusion graph* is gradually constructed by partitioning analytically the initial (four) array references using LBL *intersections* and *differences*. While the intersection of two non-disjoint LBLs is an LBL as well [33], the difference is not necessarily an LBL – and this latter operation makes the decomposition difficult. In this example, $A1 \cap A2 = A3$ and $A1 - A3$, $A2 - A3$ are also LBLs (denoted $A4$, $A5$ in Fig. 2(b)). However, the difference $A3 - A10$ is not an LBL due to the non-convexity of this set. At the end of the decomposition, the nodes without any incident arc represent non-overlapping LBLs (they are displayed in Fig. 2(c)). Every array reference in the code is now either a disjoint LBL itself (like $A10$ and $A11$), or a union of disjoint LBLs (e.g., $A1 = A4 \cup A3 = A4 \cup \bigcup_{i=6}^{11} A_i$).

When the affine vector function $\mathbf{i} \longmapsto \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$ is a one-to-one mapping, the LBL size computation reduces to the computation of the number of lattice points (i.e., having integer coordinates) in a **Z**-polytope; otherwise, it reduces [5] to counting the points in a

[1] The iterator space of an array reference is not always a convex polytope; it can be a non-convex polyhedron, or even a union of convex and nonconvex polyhedra. But, nevertheless, it can be *decomposed* into a finite set of disjoint **Z**-polytopes.



```
T[0] = 0 ;                    // A[10][529] : input
for ( j=16 ; j<=512 ; j++)
{ S[0][j-16][0] = 0 ;
  for ( k=0 ; k<=8 ; k++)
    for ( i=j-16 ; i<=j+16 ; i++)
      S[0][j-16][33*k+i-j+17] = A[4][j] - A[k][i]
                              + S[0][j-16][33*k+i-j+16] ;
  T[j-15] = S[0][j-16][297] + T[j-16] ;
}
for( j=16 ; j<=512 ; j++)
{ S[1][j-16][0] = 0 ;
  for( k=1 ; k<=9 ; k++)
    for( i=j-16 ; i<=j+16 ; i++)
      S[1][j-16][33*k+i-j-16] = A[5][j] - A[k][i]
                              + S[1][j-16][33*k+i-j-17] ;
  T[j+482] = S[1][j-16][297] + T[j+481] ;
}
out = T[994];                 // out : output
```

Figure 2: (a) *Example 2*. (b) Decomposition of the index space of signal A into disjoint linearly bounded lattices (LBLs); the arcs in the graph show the inclusion relations between LBLs. (c) The partitioning of A's array space according to the decomposition (b).

projection of a polytope [28, 40]. Counting the lattice points in a polytope can be done in several ways: there are methods based on Ehrhart polynomials like, for instance, [11, 39], or even much simpler – adapting the Fourier-Motzkin technique [12, 27]. For reason of scalability, we use a computation technique based on the decomposition of a simplicial cone into unimodular cones [6].

Step 2 *Determine the memory size at the boundaries between the blocks of code.*

After the decomposition of the array references in the specification code, a lifetime analysis on the polyhedral partitions of the signals finds the blocks of code (e.g., nest of loops) where each of the disjoint LBLs is produced and consumed (i.e., used as part of an operand for the last time). Based on this information, the memory size at the block boundaries can be computed *exactly*, since the storage requirements of LBLs are already known from *Step 1*.

Step 3 *Compute the maximum memory size inside each block.*

This operation is based on the computation of $maximum$ iterator vectors relative to the lexicographic order. Taking the set of iterator vectors mapping a given array element and assuming the loops normalized (i.e., all the iterators are increasing with the step 1), the maximum iterator vector yields the iteration in the loop nest when the element is consumed and, hence, the data storage decreases. Actually, part of the LBLs produced or consumed in the block can be conveniently ignored if their effect on the memory variation can be taken into account without generating the scalars they cover and computing their maximum iterator vectors. For instance, in the first loop nest of *Example 2* (see Fig. 2(a)), it can be proven that each iterator vector $[j\ k\ i]^T$ corresponds to a unique produced scalar $S[0][j-16][33*k+i-j+17]$ and a unique consumed scalar $S[0][j-16][33*k+i-j+16]$. The effect of the two array references on the memory variation is +1-1=0 in each iteration and, therefore, these operands can be ignored. It can be shown that the storage requirement of *Example 2* is 5,292 locations, the maximum occupancy occurring in the first loop nest. □

The memory size computation tool implementing this algorithm can be used to evaluate the loop ordering strategy described in Section 3.1.

Example 3: for (i=0; i<95; i++)
    for (j=0; j<32; j++) {
        if (i+j$\geq$ 31 && i+j$\leq$ 62) A[i][j]=$\cdots$
        if (i+j$\geq$ 63 && i+j$\leq$ 94) $\cdots$=A[i-32][j]
    }

If the A-elements are consumed in this loop nest, the minimum memory size needed by signal A is 784 locations. The trace of the memory occupancy is displayed in Fig. 3 (the first graph). In this example, i is the only spanning dimension (SD). According to the guiding principles of Section 3.1 it should hence be ordered innermost. This is substantiated by our size computation. Interchanging the loops drastically decreases the storage requirements to only 32 locations, the new trace being shown in Fig. 3, as well.

The memory size computation tool referred in this section is able to process large specifications in terms of numbers of array references, scalars, loop nests, and lines of code. More details about the theoretical model, implementation aspects and results can be found in [43].

4 Trade-offs in the Storage Exploration Methodology

Recent advanced multimedia systems use a large amount of data storage and transfers. This memory and bus usage consumes major part of the energy in the embedded system mainly due to initial bad data locality. The Data Transfer and Storage Exploration (DTSE) methodology [10] reduces the energy consumption and

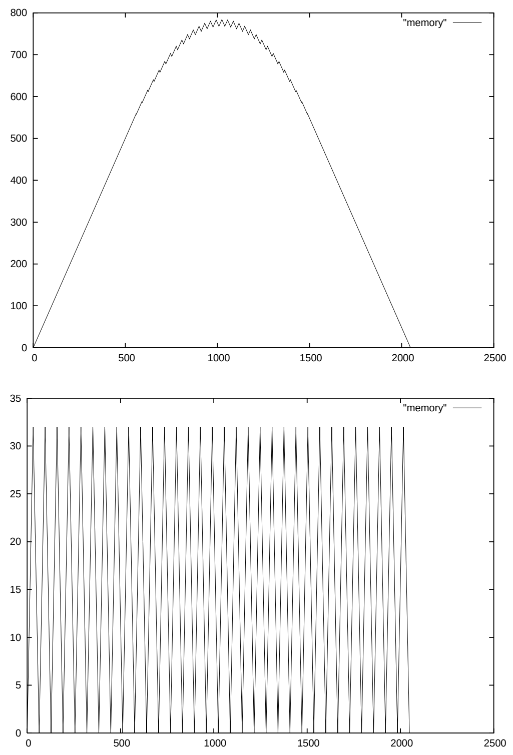


Figure 3: Memory traces for *Example 3*; the second trace shows the memory variation when the loops are interchanged.

optimizes global memory accesses by refining and improving the source code implementation.

Global Loop Transformations (GLT) are the crucial step of the DTSE methodology. They improve the initial bad data locality and thus enable subsequent optimization steps. The steps before GLT (pre-GLT) reduce redundant data transfers and expose the freedom (e.g., by inlining) for the GLT step. They trade-off data memory size vs. other parameters of the application. E.g., selective function inlining [1] creates more opportunities to data memory size reduction, however, it requires more instruction memory size. GLT step changes execution ordering of the application and thus improves locality of array accesses. This results in later data memory size reduction. However, it potentially sacrifice other parameters of the application, like the instruction memory size, control-flow complexity, number of memory accesses, etc. To evaluate these data memory size related trade-offs, we need fast and accurate storage size estimators that have been discussed in Section 3. These estimators can give us important feedback on the quality of the steering (either automatic or manual) for GLT and pre-GLT trade-offs.

Trade-offs w.r.t. the data memory size appear when we enlarge the exploration space or change execution ordering. After GLT the ordering is fixed. Still, there exist interesting trade-offs during code generation when the code is generated from the geometrical model. During this phase, we trade-off the code size, the complexity of the control-flow, and the number of empty iterations for the same geometrical model from which different codes can be generated. These trade-offs have already been discussed by the authors working on code generators from geometrical model [8, 29]. Also, they do not affect the data memory size, so we shall not discuss them further.

4.1 pre-GLT trade-offs

The GLT are performed on the geometrical model which is used in most of the research in the field of loop transformations [3, 14, 21, 22, 41]. However, the model imposes strict limitations on the input code. It can deal only with static control parts (SCoP) [7]. The static control part is a maximal set of consecutive statements without *while* loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters. Also, the geometrical model requires pointer-free code in one function [10]. The parts of the code that do not fulfill these strict conditions cannot be modeled in the geometrical model and thus cannot be transformed. To extend the exploration scope of the global loop transformations, different preprocessing techniques, like selective function inlining [1] (SFI), pointer analysis and conversion [32, 17] (PA&C), dynamic single assignment (DSA) conversion [35], hierarchical rewriting [10], and scenario creation [26], have been proposed. These techniques often require trade-offs between the freedom they allow for loop transformations and extra cost you have to pay (e.g., code size). These pre-GLT trade-offs are orthogonal to the GLT trade-offs which will be discussed in the next subsection.

4.2 GLT trade-offs

After preparing the application to be parsed to geometrical model, we can perform the GLT on this model. The GLT change the execution order of the application and affect data memory size, instruction memory size, control-flow complexity, number of memory accesses, etc.

In-place optimization reuses space in the memory of the data structure or data structure element that is not needed any more by the new-coming structure or element that is going to be produced [15]. The closer the production and the consumption of the data structure or data structure element in the program is, the better in-place optimization can be applied. Data reuse optimization means to place a local copy of the part of the array which will be used (consumed) several times closer to the data path [34]. The closer the consumptions in the program are, the better the data reuse will be. The close consumption for good data reuse can cause bad in-place optimization for some arrays and vice versa. The good data reuse causes reduction in the number of memory accesses and the bad in-place causes larger data memory size. This results to trade-off number of memory accesses and data memory size. An example of this trade-off, together with using memory size evaluation techniques from Section 3, is given in Section 4.3.

The trade-off in the previous paragraph has targeted the data part of the application. Till now we did not look at the control part of the application. The code after GLT, with optimal locality and thus minimal data memory size, contains several complex *if* conditions.

However, performance effective code should not contain complex control flow since it will slow down the application. Due to the conditions, the loops can have a problem with software pipelining. The rich control flow in the application can create overhead if good branch prediction or guarded execution is missing in the data path. The extra control flow is present because of fusion and shifting of the loops to obtain optimal locality and still to satisfy the flow dependencies. However, this control flow can be reduced by not fusing all the loops that are required for optimal locality. This will cause the growth of the memory size requirements, resulting in data memory size vs. control flow complexity trade-off.

4.3 The trade-off example

During all trade-offs discussed in the previous subsections the memory size estimation is beneficial. We shall demonstrate it on the simple example of trade-off between storage size and number of memory accesses in *Example 4*. In this example, there are three 4x5 arrays *a*, *b* and *c*, and three loop nests. In the first loop nest, arrays *a* and *b* are produced (written) resulting in 2x4x5, i.e. 40 accesses. In the second loop nest, arrays *a* and *b* are consumed (read) and array *c* is produced (written). This results in 3x4x5, i.e. 60 accesses. In the last loop nest, array *c* is consumed (read) resulting in 1x4x5, i.e. 20 accesses. Arrays *a* and *b* are still used later. Thus, the memory locations of arrays *a* and *b* cannot be reused by array *c*, i.e., arrays *a* and *b* cannot be in-placed (inter array in-place, see Section 2) with array *c*. The maximal number of simultaneously alive elements (i.e., required storage size) is 3x20, i.e. 60 (between the second and the third loop nest). The initial implementation requires thus 120 memory accesses and a storage size of 60 locations. The information about storage size can be obtained using the memory size evaluation techniques in Section 3. To obtain a better implementation, loop reverse and fusion are used. Note that it is not possible to fuse all the three loop nests because of the reverse in the second loop nest. To reverse the third loop nest is not possible due to other constraints.

Example 4:
```
int a[4][5], b[4][5], c[4][5];
for (i=0; i<4; i++)
   for (j=0; j<5; j++) {
      a[i][j] = ···
      b[i][j] = ···
   }
for (i=0; i<4; i++)
   for (j=0; j<5; j++) {
      c[3-i][4-j] = f(a[i][j],b[i][j]);
   }
for (i=2; i<6; i++)
   for (j=1; j<6; j++) {
      ··· = c[i-2][j-1];
   }
/* a & b arrays still used */
```

First we assume the fusion of the first two loop nests. Then, we can assign the computed values of arrays *a* and *b* into two temporary variables and use these variables in the *f(int, int)* function. Arrays *a* and *b* still have to be produced, because they are used later. This means the whole consumption of *a* and *b* in the second loop nest was saved, i.e., 40 accesses. Thus, the number of memory accesses has been reduced to 80. The storage size still remains 60 locations since this fusion has not affected the array lifetimes.

Another option is to fuse the last two loop nests. Before that, reverse has to be applied on the second loop nest. After the lifetime exploration of array *c* by a memory size evaluation technique from Section 3, we can determine that only some elements of this array have to be simultaneously alive. The number of these elements is computed by a memory size evaluation technique from Section 3, and is going to be 6 instead of 20. This results in an overall storage size of 46, instead of the original 60 locations. Note that the number of memory accesses does not change, only some memory locations are accessed several times after applying (intra) in-place optimizations.

To summarize, the initial implementation required a data memory size of 60 locations and 120 data memory accesses. After the loop fusion of the first two loop nests, the number of memory accesses has decreased to 80 data memory accesses. Another option is the loop fusion of the last two loop nests. This decreased the required data memory size to 46 locations. Neither solution is the best one. The fusion of the first two loop nests is better for number of memory accesses, the fusion of the last two loop nests is better for the data memory size. These two solutions represent two points on the optimal Pareto curve that trades-off the number of memory accesses and the storage size. Storage size estimators are crucial for the evaluation of the possible solutions and, also, to provide important feedback for the trade-off steering (either automatic or manual) techniques.

5 Conclusions

This paper has addressed the important role of loop transformations for enhancing the memory management of signal processing systems, whose behavior is described by high-level, array-oriented specifications. In this context, the paper has discussed basic memory management trade-offs taken into account during the exploration of the design space.

References

[1] J.Absar, F.Catthoor, K.Das, “Call-instance based function inlining for increasing data access related optimization opportunities,” *Technical report*, IMEC, Leuven, Belgium, 2003.

[2] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, USA: Kluwer Acad. Publ., 1988.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, D. Padua, “Automatic program parallelization”, *Proc. of the IEEE*, vol.81, no.2, pp.211-243, Feb. 1993.

[4] F. Balasa, F. Catthoor, H. De Man, “Background memory area estimation for multi-dimensional signal processing systems,” *IEEE Trans. VLSI Syst.*, vol. 3, no. 2, pp. 157-172, June 1995.

[5] F. Balasa, F. Catthoor, H. De Man, ‘Practical solutions for counting scalars and dependences in ATOMIUM – a memory management system for multi-dimensional signal processing,” *IEEE Trans. CAD of IC’s and Syst.*, vol. 16, no. 2, pp. 133-145, Feb. 1997.

[6] A.I. Barvinok, “A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed,” *Math. of Operations Research*, vol. 19, no. 4, pp. 769-779, Nov. 1994.

[7] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, O. Temam, “Putting polyhedral loop transformations to work,” *Proc. Int. Workshop Languages & Compilers for Parallel Comput.*, pp. 209-225, Sept. 2003.

[8] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” *Proc. Int. Conf. on Parallel Arch. and Compilation Techniques*, pp. 7-16, Sept. 2004.

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

[10] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Boston, USA: Kluwer Acad. Publ., 2002.

[11] Ph. Clauss, V. Loechner, “Parametric analysis of polyhedral iteration spaces,” *J. VLSI Signal Processing*, vol. 19, no. 2, pp. 179-194, 1998.

[12] G.B. Dantzig, B.C. Eaves, “Fourier-Motzkin elimination and its dual,” *J. Combinatorial Theory (A)*, vol. 14, pp. 288-297, 1973.

[13] A. Darte, R. Schreiber, G. Villard, “Lattice-based memory allocation,” *IEEE Trans. Computers*, vol. 54, pp. 1242–1257, Oct. 2005.

[14] A. Darte, Y. Robert, “Affine-by-statement scheduling of uniform and affine loop nests over parametric domains”, *J. Parallel and Distributed Computing*, vol. 29, no. 1, pp. 43-59, 1995.

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

[16] P. Feautrier, “Dataflow analysis of array and scalar references,” *Int. J. Parallel Programming*, vol. 20, no. 1, pp. 23-52, 1991.

[17] B. Franke, M. O’Boyle, “Array recovery and high-level transformations for DSP applications”, *ACM Trans. Embedded Computing Syst.*, vol. 2, no. 2, pp. 132-162, May 2003.

[18] D.Gajski, F.Vahid, S.Narayan, J.Gong, *Specification and design of embedded systems*, Prentice Hall, Englewood Cliffs NJ, 1994.

[19] P. Grun, F. Balasa, N. Dutt, “Memory size estimation for multimedia applications,” *Proc. 6th Int. Workshop Hardware/Software Co-Design*, pp. 145-149, Mar. 1998.

[20] Q. Hu, A. Vandecappelle, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, “Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications,” *Proc. Asia & S.-Pacific Design Automation Conf.*, pp. 606–611, Jan. 2006.

[21] M. Kandemir, J. Ramanujam, A. Choudhary, P. Banerjee, “A layout-conscious iteration space transformation technique,” *IEEE Trans. on Computers*, vol. 50, no. 12, pp. 1321-1335, 2001.

[22] W. Kelly, W. Pugh, “A framework for unifying reordering transformations,” Univ. Maryland College Park, CS-TR-3193, Apr. 1993.

[23] P.G. Kjeldsberg, F. Catthoor, and E. J. Aas, “Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications,” *Proc. 38th ACM/IEEE Design Automation Conf.*, pp. 365–370, June 2001.

[24] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, “Data dependency size estimation for use in memory optimization,” *IEEE Trans. CAD of IC’s and Syst.*, vol. 22, no. 7, pp. 908-921, July 2003.

[25] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, “Storage requirement estimation for optimized design of data intensive applications,” *ACM Trans. Design Aut. Electronic Syst.*, vol. 9, pp. 133–158, Apr. 2004.

[26] M. Palkovic, H. Corporaal, F. Catthoor, “Dealing with data dependent conditions to enable general global source code transformations”, to appear in *Int. J. of Embedded Syst.*, vol. 2-3, June 2006.

[27] W. Pugh, “A practical algorithm for exact array dependence analysis,” *Comm. of the ACM*, vol. 35, no. 8, pp. 102-114, Aug. 1992.

[28] W. Pugh and D. Wonnacott, “An exact method for analysis of value-based array data dependences,” *Proc. 6th Int. Workshop Languages and Compilers for Parallel Computing*, pp. 546–566, Aug. 1993.

[29] F. Quillere, S. Rajopadhye, D. Wilde, “Generation of efficient nested loops from polyhedra”, *Int. J. Parallel Programming*, vol. 28, no. 5, Oct. 2000.

[30] J. Ramanujam, J. Hong, M. Kandemir, A. Narayan, “Reducing memory requirements of nested loops for embedded systems,” *Proc. 38th ACM/IEEE Design Automation Conf.*, pp. 359-364, June 2001.

[31] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.

[32] L. Semeria, G. De Micheli, “SpC: synthesis of pointers in C”, *Proc. IEEE Int. Conf. CAD*, pp. 340-346, Nov. 1998.

[33] L. Thiele, “Compiler techniques for massive parallel architectures,” in *State-of-the-art in Computer Science* (ed. P. Dewilde), Kluwer Acad. Publ., 1992.

[34] T. Van Achteren, G. Deconinck, F. Catthoor, R. Lauwereins, “Data reuse exploration methodology for loop-dominated applications”, *Proc. 5th ACM/IEEE Design and Test in Europe Conf.*, pp. 428-435, April 2002.

[35] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, F. Catthoor, “Advanced copy propagation for arrays,” *Proc. SIGPLAN Conf. Languages, Compilers, and Tools for Embedded Syst,*, pp. 24-33, June 2003.

[36] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, “Transformation to dynamic single assignment using a simple data flow analysis,” *Proc. 3rd Asian Symp. on Programming Languages and Syst.*, vol. 3780 of *Lecture Notes Comp. Sc.*, pp. 330–346, Springer Verlag, Nov. 2005.

[37] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man, “Background memory management for the synthesis of algebraic algorithms on multi-processor dsp chips,” *Proc. Int. Conf. on VLSI*, pp. 209–218, Aug. 1989.

[38] I. Verbauwhede, C. Scheers, J.M. Rabaey, “Memory estimation for high level synthesis,” *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 143-148, June 1994.

[39] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, M. Bruynooghe, “Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations,” *Proc. Int. Conf. Compilers Arch. and Synthesis for Embedded Syst.*, pp. 248-258, Sept. 2004.

[40] S. Verdoolaege, K. Beyls, M. Bruynooghe, F. Catthoor, “Experiences with enumeration of integer projections of parametric polytopes,” in *Compiler Construction: 14th Int. Conf.* (ed. R. Bodik), vol. 3443, pp. 91-105, Springer, 2005.

[41] M.E. Wolf, M.S. Lam, “A data locality optimizing algorithm”, *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 30-43, June 1991.

[42] Y. Zhao, S. Malik, “Exact memory size estimation for array computations,” *IEEE Trans. VLSI Syst.*, vol. 8, no. 5, pp. 517-521, 2000.

[43] H. Zhu, I.I. Luican, F. Balasa, “Memory size computation for multimedia processing applications,” *Proc. Asia & South-Pacific Design Automation Conf.*, pp. 802-807, Jan. 2006.

[44] IMEC, “Atomium web site,” http://www.imec.be/design/atomium/ .