Storage Estimation and Design Space Exploration Methodologies for the Memory Management of Signal Processing Applications*

F. Balasa* P.G. Kjeldsberg† A. Vandecappelle‡ M. Palkovic‡ Q. Hu† H. Zhu* F. Catthoor§

* Dept. of Computer Science, University of Illinois at Chicago, Chicago, U.S.A. {fbalasa,hzhu7}@uic.edu
† Norwegian University of Science and Technology, Trondheim, Norway {pgk,hu}@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-dominated 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. This paper gives a tutorial overview on the existing techniques for the evaluation of the data memory size, which is an important step during the early stage of system-level exploration. The paper focuses on the most advanced developments in the field, presenting in more detail (1) an estimation approach for non-procedural specifications, where the reordering of the loop execution within loop nests can yield significant memory savings, and (2) an exact computation approach for procedural specifications, with relevant memory management applications – like, measuring the impact of loop transformations on the data storage, or analyzing the performance of different signal-to-memory mapping models. Moreover, the paper discusses typical memory management trade-offs – like, for instance, between storage requirement and number of memory accesses – taken into account during the exploration of the design space by loop transformations in the system specification.

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 [1], [2], [3].

The behavior of these targeted VLSI systems, synthesized to execute mainly data-dominated applications, is usually described in a high-level programming language. The code is typically organized in sequences of loop nests having as boundaries (typically, affine) functions of the loop iterators. The code may contain conditional instructions, where the arguments can 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 [1].

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

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 opportunities [4], [5], [6]. If these data are mapped to the intermediate layer memories in the hierarchy, the off-chip memory accesses are potentially significantly reduced, which is critical for system performance and energy consumption [2]. But this is heavily enabled by the in-place mapping of the data copies in the intermediate memory, which is usually quite size-constrained [7]. Hence, size reduction of the data storage is also crucial for this purpose. Several lower-level techniques are available for such a size optimization but they are too slow to use during the actual system-level exploration stage, up front. For that purpose, we need efficient system-level evaluation or estimation techniques.

This paper provides a tutorial overview on the existing techniques for the system-level evaluation of the data memory size in signal processing applications, focusing on the most advanced developments in this research domain. The paper presents in more detail (but, still, at high level) two more recent, complementary approaches for the evaluation of the memory size of *non-procedural* and *procedural* algorithmic specifications. The first technique – developed at the Norwegian University of Science and Technology (NTNU), Trondheim, Norway, with the co-operation of the Interuniversity Microelectronics Center (IMEC), Leuven, Belgium – performs an estimation of the data memory size, where the reordering of the loop execution within loop nests is optimized such that the storage requirements be reduced. The second technique – developed at the University of Illinois at Chicago (UIC), U.S.A., with initial support from IMEC – performs an exact computation of the minimum data storage for procedural specifications. This latter approach has relevant memory management applications like the study of the impact of loop transformations on storage requirements and the performance analysis of different models of mapping multi-dimensional signals to the physical memories. Moreover, the paper discusses typical memory management trade-offs – extensively studied at IMEC (like, for instance, trading the number of memory accesses for the size of the data memory), that are taken into account during the exploration – mainly, by loop transformations – of system specifications. Being a tutorial presentation, the paper does not address lower-level computational aspects (a rich bibliography is offered to the interested reader); instead, it focuses on the general computation models of the approaches, on their applications in the memory management and system-level exploration methodologies, illustrating the concepts and ideas by several code examples.

The rest of the paper is organized as follows. After an overview of the past works addressing the memory estimation/computation problem (Section 2), the paper presents in Section 3 two more 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. Section 4 discusses memory management trade-offs that must be taken into account during the early high-level design space exploration. Section 5 presents several experimental results and Section 6 states the main conclusions of this comprehensive overview.

2 The Memory Size Computation Problem: A Brief Overview

For several decades, researchers have worked on different approaches for estimating or computing the minimum memory size required to execute a given application. Most of the initial work was done at scalar level due to the register-transfer behavioral specifications of the earlier digital systems. After being modeled as clique partitioning problem [8], the register allocation and assignment have been optimally solved for nonrepetitive schedules, when the life-time of all the scalars is fully determined [9]. The similarity with the problem of routing channels without vertical constraints [11] has been exploited in order to determine the minimum register requirements (similar to the number of tracks in a channel), and to optimally assign the scalars to registers (similar to the assignment of one-segment wires to tracks) in polynomial time by using the *left-edge* algorithm [10]. A nonoptimal extension for repetitive and conditional schedules has been proposed in [12]. A lower bound on the register cost can be found at any stage of the scheduling process using force-directed scheduling [13]. Integer Linear Programming techniques are used in [14] to find the optimal number of memory locations during a simultaneous scheduling and allocation of functional units, registers, and busses. Employing circular graphs, [15] and [16] proposed optimal register allocation/assignment solutions for repetitive schedules. A lower bound for the register count is found in [17] without fixing the schedule, through the use of ASAP and ALAP constraints on the operations. A good overview of these techniques can be found in [18].

Common to all the scalar-based techniques is that they break down when used by flattening large multi-dimensional arrays, each array element being considered a separate scalar. The nowadays data-intensive signal processing applications are described by high-level, loop-organized, algorithmic specifications whose main data structures are typically multi-dimensional arrays. Flattening the arrays from the specification of a video or image processing application would typically result in many thousands or even millions of scalars.

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-based methodology. We will now present different published contributions using this approach, starting with techniques that assume a procedural execution of the application code.

In [19], 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 the internal in-place mapping of an array (intra-array in-place) is considered; the possibility of mapping arrays in-place of each other (inter-array in-place) is not exploited. By in-place mapping we mean the optimization technique where data with non-overlapping lifetimes can be mapped to the same physical memory locations [20].

Another approach is taken in [21]. 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 [22] 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 [23], the specifications are limited only to perfectly nested loops. A reference window is used for each array. At any moment 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 the local memory. The maximal window size 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. [5] 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 [24] 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 [25]. The maximal combined size of simultaneously alive basic sets found through a greedy graph traversal gives an estimation of 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. Optimizations at these early steps are also the main topic of this paper. In addition to this, much effort has been focused on performing the final in-place optimization and signal-to-memory mapping. A good overview of the work in this field can be found in [26]. Although this paper will not put much emphasis on the later design step of assigning signals to the physical memory, this mapping problem will be briefly addressed in Section 3.2.3, as an application of one of the memory size computation approaches. The next section will present two alternative techniques that can be used for the system-level evaluation of the data storage.

3 Non-Scalar Models for Memory Size Evaluation

This section presents two approaches for the memory size evaluation of loop-organized algorithmic specifications, where the main data structures are multi-dimensional arrays. The specifications are in the *single-assignment* form, that is, each array element is written at most once (but it can be read an arbitrary number of times).

The first technique assumes that the application code can be interpreted in a non-procedural form. It allows any degree of fixation of the execution ordering, from fully unfixed, through partially fixed, to fully fixed. To be able to quickly generate estimates with this limited amount of information available, certain approximations are used. Unless the ordering is fully fixed, it also generates upper and lower bounds on the storage requirement. It is, therefore, natural to use this approach during the early system level design steps. The designer can use the bounds as guidance when making decisions that gradually fix the execution ordering.

The second approach assumes that the application code is written in a procedural form. It is, therefore, beneficial to be used at later design stages when it makes sense to allow more computationally hard techniques in order to reach exact results.

3.1 Estimation Approach for Non-Procedural Specifications

All but the last estimation approach described in Section 2 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 [3]. The number of alternative solutions is huge, each with a different execution ordering. It is very time consuming to evaluate each of these using its fixed ordering. The designer would benefit from having size estimates that work with only a partially fixed execution ordering. In that context, an estimate that is not necessarily fully exact is acceptable. Actually, without the ordering being fixed, it is even infeasible to come up with one number for the size: in practice, a range of sizes is possible still. So, we would like to compute as tight as possible bounds on that available range for a given amount of freedom in the loop organization.

For our target classes of data-dominated 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* [27]. In this example the array index expressions are relatively simple, but the techniques presented in this section work for any affine and manifest index expressions. Section 4.1 discusses preprocessing techniques that can be employed on the original source code, if this is not the case.

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>=z) B[x][y][z]=f2(A[x-1][y-z][z]) ;
    }
```

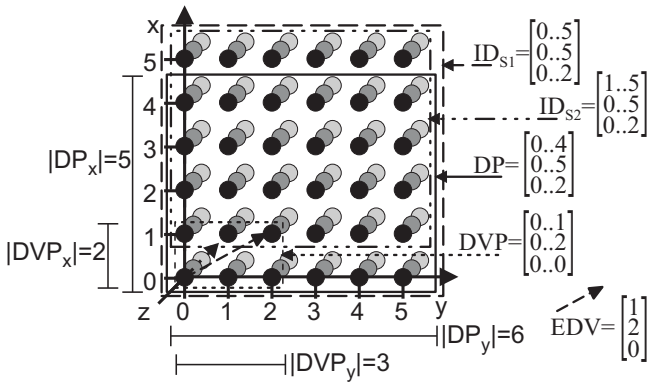


Figure 1: Iteration space and iteration domains (IDs) from *Example 1*.

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 8. 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 shall 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 data dependency information from application code.*

Our estimation methodology uses a geometrical model to describe data, operations and dependencies. The first part of this step is performed by the Atomium tool [28]. It finds the *Iteration Domains* (ID) of the statements and the index expressions of the array accesses in the application code. Fig. 1 shows graphically the *iteration space* [27] of the loop nest in *Example 1* with the IDs of the two statements. To avoid unnecessarily complex figures, two-dimensional rectangles are used. All nodes within a rectangle are part of the corresponding ID. For our example, the ID of *S2* has originally a somewhat complex shape because of the if-clause $y >= z$. When this is the case, we apply a bounding box simplification. A bounding box iteration domain is a rectangular approximation of the original iteration domain, where the original can have any convex shape. Fig. 1 shows ID_{S2} after this simplification. This is an efficient and appropriate approximation since most convex shapes in our targeted application domain are rectangular and have regular accesses, i.e., images, blocks, etc. To avoid too complex address generation, linear addresses are also normally used in the final implementation of the application. Bounding box based size estimates will then give results similar to those of the final implementation.

At each *iteration node* within ID_{S1} and ID_{S2} we assume that statement *S1* and *S2* are executed, respectively. For ID_{S1} this is fully correct, while it is an approximation for ID_{S2} because of the bounding box. Not all elements produced by *S1* 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. Note that as long as no data dependencies are violated, i.e., data is never consumed before it is produced, we can visit the iteration nodes in any order we like. This

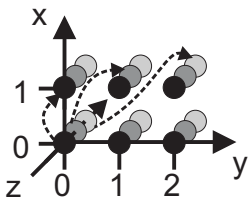


Figure 2: Dependency vectors (DVs) in *Example 1*.

corresponds to a non-procedural interpretation of the application code.

We now need to find the *Extreme Dependency Vector* (EDV) for each dependency. For this we can use an efficient technique introduced by Hu *et al.* in [29] that avoids the traditional computationally expensive ILP-based solutions. There is a *Dependency Vector* (DV) from each iteration node writing an array element to the iteration node reading the same array element. Dependencies are uniform if they all have the same length and direction. Otherwise, they are nonuniform, as in *Example 1*. Fig. 2 summarizes the three alternative DV lengths and directions of *Example 1*. All DVs have length 0 in the z-dimension. The EDV is the maximal projection among all DVs of each loop dimension. In most cases, as in *Example 1*, the result of this is one of the actual DVs. If this is not the case, use of the EDV will result in some overestimates, as shown in [29]. Alternatively, these exceptions can be handled separately during estimation. In [29], a concept of a *Maximal Dependency Vector* is introduced that is always one of the existing DVs. To be able to use this, a fully fixed execution ordering is required. This is consequently not an alternative here.

The EDV spans a *Dependency Vector Polytope* (DVP), as shown in Fig. 1. Its dimensions are defined as *Spanning Dimensions* (SD). The remaining dimensions are denoted *Nonspanning Dimensions* (ND). In Fig. 1, *x* and *y* are SDs while *z* is ND. More details about these concepts can be found in [30].

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 [31]. A best case and worst case common iteration space may be necessary to find the memory size LB and UB for the complete application.

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 $LR_y = 2/5$. 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. Table 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 [30].

	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= $Q_x \cdot P_y \cdot P_z$=18
2nd nest level	LB_t=LB_t+$Q_x \cdot P_y$=6 UB_t=UB_t+$Q_x \cdot P_z$=33	LB_t=LB_t=18 UB_t=UB_t+$Q_y \cdot P_z$=24
3rd nest level	LB=LB_t+Q_y=8 UB=UB_t=33	LB=LB_t+Q_y=20 UB=UB_t=24

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 [32].

We will demonstrate the benefit of allowing a partially fixed ordering using a part of the Updating Singular Value Decomposition (USVD) algorithm [33] used, for instance, in beamforming for antenna systems. The part we investigate has two major dependencies, each with estimated LB/UB of 1/100 if no execution ordering is fixed. For one of these dependencies there exist restrictions in the code, forcing a given dimension to be fixed outermost. Estimating the sizes with this partially fixed ordering results in converging LB/UB of 100/100 for both dependencies. Both are alive simultaneously, so their combined size is 200. This is found without having to explore all the alternative orderings. The large penalty of fixing this dimension outermost would encourage the designer to investigate transformation alternatives. One possibility would be to perform a loop body split, so that the dependency without any restrictions could be ordered optimally. This would then result in a combined size of 101. Sections 4 and 5 give more detailed and larger examples of how this estimation methodology can be a part of a solution space exploration during the global loop transformation design step.

3.2 Exact Computation Approach for Procedural Specifications

This section presents a non-scalar method for computing *exactly* the minimum data 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 (minimum data memory size) 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 [4], [5], [6] and, therefore, a better steering of the (hierarchical) memory allocation [1], [2]. The trade-off will be the longer run-times and the requirement that each possible loop transformation instantiation has to be evaluated individually. That makes this approach better suited after the crude pruning of the system-level data-flow, and the loop transformation stage has been finalized, so that only a limited set of options remains.

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 $\mathbf{Z}$-polytope (its iterator space) $\{ \mathbf{i} \in \mathbf{Z}^n \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b} \}$, therefore, a *lattice* [34] which is *linearly bounded* [35]. For instance, the array reference $A[2i - j + 5][3i + 2j - 7]$ from the loop nest

```
for (i=2; i≤7; i++)
   for (j=1; j≤-2i+15; j++)
       if (j≤i+1)  ···  A[2i-j+5][3i+2j-7] ···
```

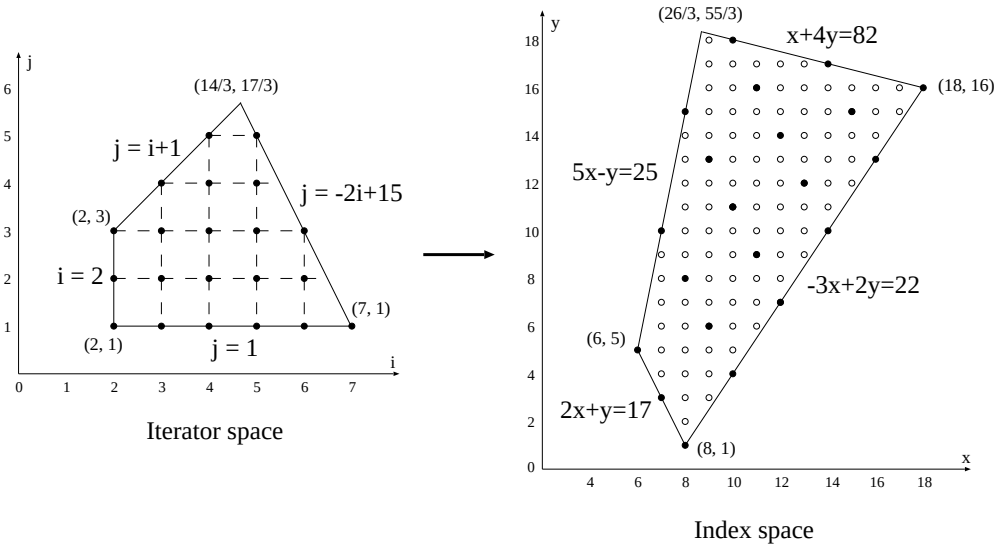


Figure 3: The iterator space and the index space of the array reference $A[2i-j+5][3i+2j-7]$.

has the iterator space $P = \left\{ \left[\begin{array}{c} i \\ j \end{array} \right] \in \mathbf{Z}^2 \;\middle|\; \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -2 & -1 \end{array} \right] \left[\begin{array}{c} i \\ j \end{array} \right] \geq \left[\begin{array}{c} 2 \\ 1 \\ -1 \\ -15 \end{array} \right] \right\}$. (The inequality $i \leq 7$ is redundant.)

The A-elements of the array reference have the indices x, y: $\left\{ \left[\begin{array}{c} x \\ y \end{array} \right] = \left[\begin{array}{cc} 2 & -1 \\ 3 & 2 \end{array} \right] \left[\begin{array}{c} i \\ j \end{array} \right] + \left[\begin{array}{c} 5 \\ -7 \end{array} \right] \middle| \left[\begin{array}{c} i \\ j \end{array} \right] \in P \right\}$.

The points of the index space lie inside the $\mathbf{Z}$-polytope $\{\ 2x + y \geq 17\ ,\ 5x - y \geq 25\ ,\ 3x - 2y \leq 22\ ,\ x + 4y \leq 82\ ,\ \ x, y \in \mathbf{Z}\}$, whose boundary is the image of the boundary of the iterator space P (see Fig. 3). However, only the points (x,y) satisfying also the divisibility condition $7 \mid 2x + y + 4$ (that is, 7 divides exactly 2x+y+4) belong to the index space; these are the black points in the right quadrilateral from Fig. 3. In this illustrative example, each point in the iterator space is mapped to a distinct point of the index space, but this is not always the case.

3.2.1 The algorithm computing the minimum data memory size

The main steps of the memory size computation algorithm [36], [37] 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 4(b) shows the result of this decomposition for the 2-dimensional signal A in the illustrative example from Fig. 4(a). The graph displays the inclusion relations ($arcs$) between the lattices of A ($nodes$). The 4 "bold" nodes are the 4 array references of signal A in the code. (The delay operator "@" indicates inputs or signals produced in previous data-sample executions of the code, and the following argument signifies the *number* of such previous executions. The delayed array references do not affect this decomposition step, but the next steps take into account the *delayed* signals since they must be kept alive during several *time* iterations.) 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 *intersections* and *differences* of lattices [37]. While the intersection of two non-disjoint LBLs is an LBL as well [35], the difference is not necessarily

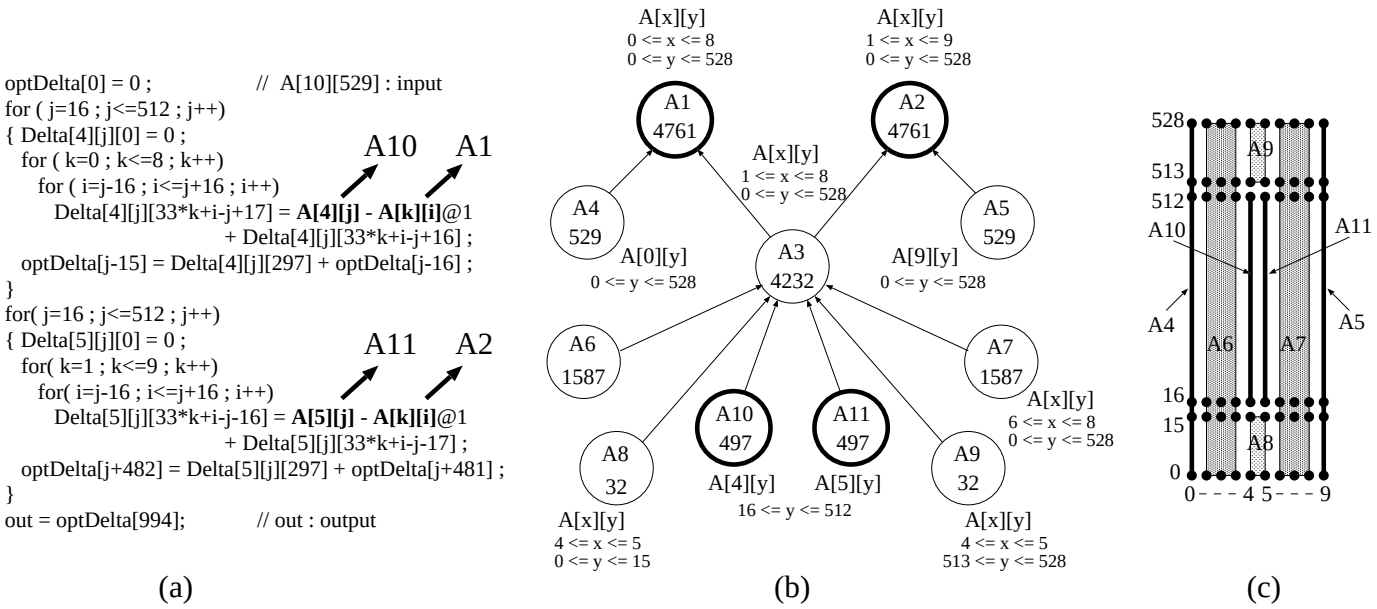


Figure 4: (a) *Example 2* (the kernel of a motion detection algorithm [1], where $m = 4$, $n = 16$, $M = 5$, $N = 512$). (b) Decomposition of the index space of signal A into disjoint linearly bounded lattices (LBLs); the arcs in the graph show the inclusion relations between LBLs. (c) The partitioning of A's array space according to the decomposition (b).

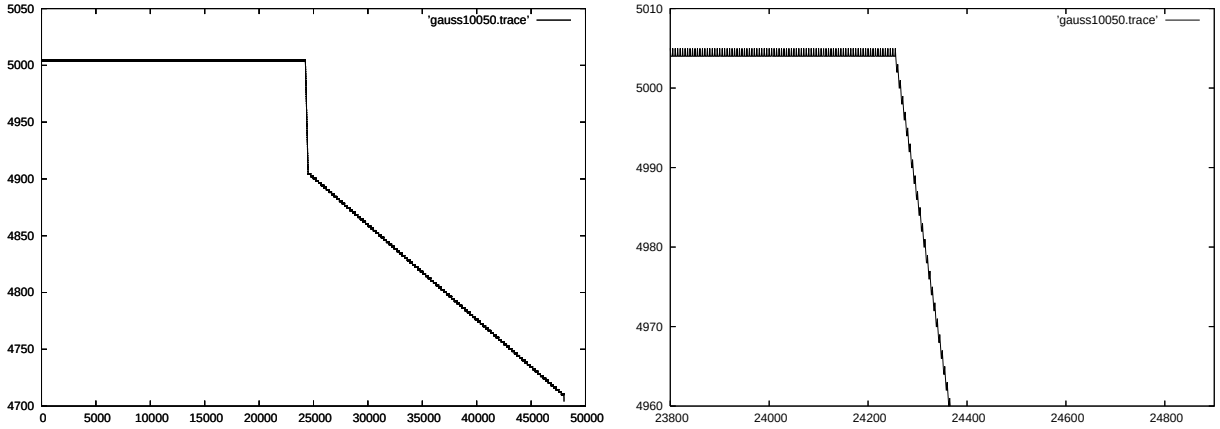
an LBL – and this latter operation makes the decomposition difficult. In this example, $A1 \cap A2 = A3$ and $A1 - A3$, $A2 - A3$ are also LBLs (denoted $A4$, $A5$ in Fig. 4(b)). However, the difference $A3 - A10$ is not an LBL due to the non-convexity of this set. At the end of the decomposition, the nodes without any incident arc represent non-overlapping LBLs (they are displayed in Fig. 4(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[1] (like in Fig. 3, where each of the 21 iterator vectors – represented by black points in the iterator space – is mapped to a distinct point in the index space), the LBL size computation reduces to the computation of the number of lattice points (i.e., having integer coordinates) in a **Z**-polytope. (See again Fig. 3, where instead of counting the black points in the right quadrilateral containing holes, we count the black points in the left quadrilateral without any hole if loop normalization was performed in advance.) Otherwise, the problem reduces to counting the points in a projection of a polytope [38], [39], as explained and exemplified in [25]. Counting the lattice points in a polytope can be done in several ways: there are methods based on Ehrhart polynomials like, for instance, [40], [41], or even much simpler – adapting the Fourier-Motzkin technique [42], [43]. For reason of scalability, we use a computation technique based on the decomposition of a simplicial cone into unimodular cones [44], approach exemplified in [36].

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 lattices 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

[1] This occurs, for instance, when the rank of matrix **T** is equal to the number of its columns, as proven in [25].



(a) (b)

Figure 5: (a) Memory trace of a 2-D Gaussian blur filter algorithm ($N = 100$, $M = 50$). The global maximum is at the point (x=5, y=5,005), therefore the minimum data memory size is 5,005 locations. (b) A detail of the memory trace where the "horizontal Gaussian blur" ends and the "vertical Gaussian blur" begins.

boundaries can be computed *exactly*, since the storage requirements of LBLs are already known from *Step 1*.

Step 3 *Compute the maximum storage requirement 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. 4(a)), it can be proven that each iterator vector $[j\ k\ i]^T$ corresponds to a unique produced scalar $Delta[4][j][33 * k + i - j + 17]$ and a unique consumed scalar $Delta[4][j][33 * k + i - j + 16]$. 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 10,582 locations, the maximum occupancy occurring in the first loop nest. □

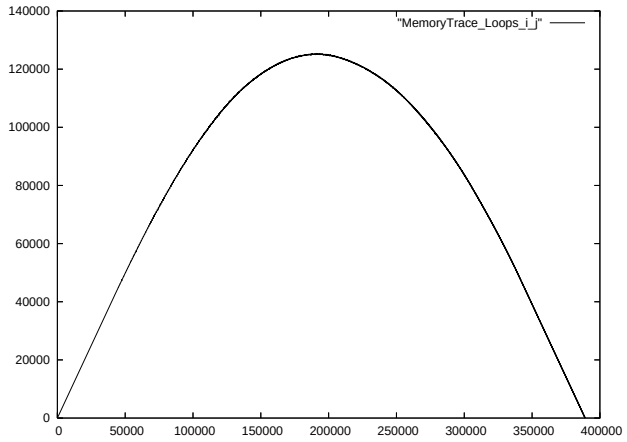
3.2.2 Evaluating the impact of loop transformations on the data storage

The tool implemented based on the algorithm computing the minimum data storage – described in Section 3.2.1 – can be easily adapted to generate the data memory variation during the execution of the application code. The memory trace generated in Fig. 5 shows the variation of the storage during the execution of a 2-D Gaussian blur filter algorithm from a medical image processing application which extracts contours from tomograph images in order to detect brain tumors. The abscissae are the numbers of datapath instructions in the code; the ordinates are the memory locations in use. While the first graph represents the entire trace, the second graph is a detailed trace in the interval [23800, 24900], which corresponds to the end of the "horizontal Gaussian blur" and the start of the "vertical Gaussian blur."

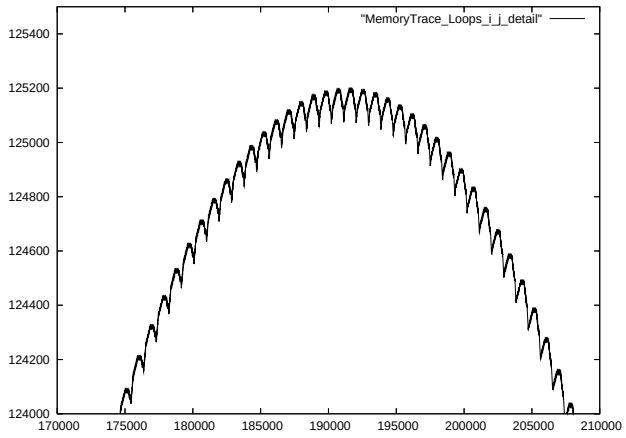
The computation of the minimum data storage is also useful in evaluating the impact of different code (and, in

```
                                  //  All the array elements are produced
for ( i=0; i<767; i++ )           //        and consumed in this loop nest
  for ( j=0; j<256; j++ ) {
    if ( i+j>=127 && i+j<=254 && j<=127 )  A[i][j] = ... ;
     if ( i+j>=255 && i+j<=382 && j<=127 )  ... = A[i-128][j];
     if ( i+j>=159 && i+j<=318 && j<=159 )  B[i][j] = ... ;
     if ( i+j>=319 && i+j<=478 && j<=159 )  ... = B[i-160][j];
     if ( i+j>=191 && i+j<=382 && j<=191 )  C[i][j] = ... ;
     if ( i+j>=383 && i+j<=574 && j<=191 )  ... = C[i-192][j];
     if ( i+j>=223 && i+j<=446 && j<=223 )  D[i][j] = ... ;
     if ( i+j>=447 && i+j<=670 && j<=223 )  ... = D[i-224][j];
     if ( i+j>=255 && i+j<=510 )                E[i][j] = ... ;
     if ( i+j>=511 && i+j<=766 )                ... = E[i-256][j];
  }
```

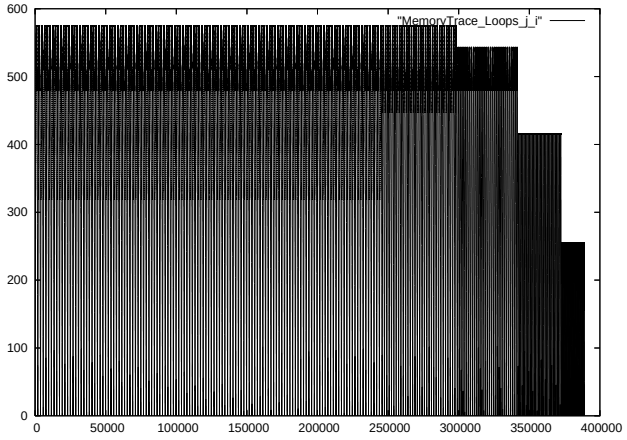
(a)



(b)



(c)



(d)

Figure 6: (a) *Example 3*. Memory traces showing the effect of loop interchange on the storage requirement: (b) The memory trace of the code. The minimum data storage (corresponding to the *maximum* value of the memory variation) is 125,193 locations. (c) A detail of the memory trace in a neighborhood of the global maximum. (d) Memory trace of the code after loop interchange. The minimum data storage is only 576 locations.

particular, loop) transformations on the data storage. For instance, the minimum memory size needed for the execution of *Example 3* from Fig. 6(a) is 125,193 locations – as computed by the software tool. The trace of the memory variation is displayed in Fig. 6(b) and a detail containing the global maximum is shown in Fig. 6(c). Different variants of code of a same application can be compared one against another in storage point of view, without the need of performing a proper memory allocation for each variant – a significantly more expensive solution. In particular, the memory size computation tool implementing the algorithm in Section 3.2.1 can be used to evaluate the loop ordering strategy described in Section 3.1. In *Example 3*, i is the only spanning dimension (SD). According to the guiding principles of Section 3.1, the loops should hence be ordered with the i dimension innermost. This is substantiated by our size computation. By interchanging the loops, the storage requirement decreases drastically with over 99.5% (to only 576 locations), the new trace being the graph shown in Fig. 6(d).

Example 3	Array A	Array B	Array C	Array D	Array E	Total
Number of array elements	32,640	51,040	73,536	100,128	130,816	388,160
Minimum array windows	12,352	19,280	27,744	37,744	49,280	146,400
Mapping array windows	16,384	25,600	36,864	50,176	65,536	194,560

Table 2: Evaluation of a signal-to-memory mapping model [45] for *Example 3* (Fig. 6(a)).

3.2.3 Mapping multi-dimensional arrays into the data memory

The minimum data storage represents the *tight lower bound* for which the execution of the code is still possible. However, in practice, this amount of storage is difficult to reach (although still possible!) since it would require a complex hardware for address generation. Instead, industrial 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. Academic researchers have studied the mapping models as well, aiming to find better trade-offs. These mapping models actually trade-off an excess of storage for a less complex address generation hardware.

Several signal-to-memory mapping techniques have been proposed (an overview is given, for instance, in [26]), but none of these research works could provide consistent information on *how effective the mapping models are*, that is, how large is the oversize of their resulting storage amount after mapping in comparison to the minimum data storage. The effectiveness of the mapping models are assessed only *relatively*, comparing the storage resulted after applying the mapping model either to the total number of array elements, or to the storage results when applying a different mapping model [45], [46], [26]. This *relative* evaluation is not sufficient though, since it does not give a precise picture on the *absolute* quality of the model.

The computation approach described in Section 3.2.1 can determine not only the minimum data storage, but, in addition, it can find out the minimum windows for each array in the application code, that is, the maximum number of each array's elements that are simultaneously alive. For instance, the signal A from *Example 3* in Fig. 6(a) needs a minimum window of 12,352 memory locations since there are at most 12,352 A-elements simultaneously alive (as computed by the tool based on the algorithm presented in Section 3.2.1). The minimum windows (or the optimal *intra-array* in-place mapping [1]) of all the signals in *Example 3*, together with the number of array elements in a static allocation, are shown in Table 2. However, since the elements of the arrays $A, \ldots, E$ can share the same locations if their lifetimes are disjoint, the minimum storage requirement is, actually, 125,193 locations, an amount smaller than the sum of the minimum windows of the five arrays in the code – that is, 146,400. This is also called the optimal *inter-array* in-place mapping [1].

To illustrate the analysis of effectiveness of a mapping approach [47], let us consider the mapping model proposed in [45] that computes storage windows for each array in the code, investigating all the possible linearizations of the array; for each linearization, the largest distance between two live elements is computed. This distance plus 1 is the data storage allocated for the array – according to [45]. For instance, the linearizations considered for a 2-D array are the ones obtained by concatenating the rows, or concatenating the columns, in the increasing or decreasing order of the indexes. When applied to the code *Example 3*, the mapping model [45] yields a storage window of 16,384 locations since, e.g., the elements $A[0][127]$ and $A[128][126]$ are simultaneously alive, and their distance in the row-by-row concatenation is

128×128-1=16,383. Therefore, an allocation steered by this model would require for signal A 32.64% more storage than really necessary (i.e., 12,352). All the array windows yielded by the mapping model [45] are displayed in the last row of Table 2. In conclusion, the model [45] would allocate 194,560 memory locations for the entire code, therefore, 32.9% more storage than the optimal memory sharing within each array (146,400 locations) and 55.41% more storage than the absolute minimum requirement (the overall optimal sharing) of 125,193 locations.

Note that, according to the estimation approach described in Section 3.1, the storage requirement of the procedural code in *Example 3* is 194,560 memory locations, which is exactly the result obtained above after the array mapping when DeGreef's model [45] was applied. The explanation is that the estimation approach is anticipating the memory allocation, it does not attempt to minimize the data memory. So, the two approaches in Sections 3.1 and 3.2 are actually complementary to each other.

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 parts of the energy in an embedded system mainly due to initially bad data locality. The systematic Data Transfer and Storage Exploration (DTSE) methodology [2] reduces the energy consumption and optimizes global memory accesses by refining and improving the source code implementation.

Global Loop Transformations (GLT) form a major step in this 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. For instance, selective function inlining [48] creates more opportunities to data memory size reduction. However, it increases the required instruction memory size. The GLT step changes the global execution ordering of the application and thus improves locality of the 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 the fast and accurate storage size estimators that have been discussed in Section 3. These estimators can yield important feedback on the quality of the steering (either automatic or manual) for both the GLT and pre-GLT trade-offs. This is shown in Fig. 7 where the trade-off oriented loop transformation structural outline and the relation to memory size estimation is presented.

In the flow chart in Fig. 7 we distinguish four types of trade-offs: *pre-GLT*, *GLT*, *Other Loop Transformations (LT)*, and *Code generation trade-offs*. These four categories of trade-offs appear in the order in which they occur in the DTSE methodology [2]. The relation between memory size estimation and those trade-offs is depicted by the arrows pointing to the trade-offs. A solid arrow means a strong relation, i.e., the memory size estimation is an important part of that trade-off. The dashed arrow depicts a weak relation, i.e., the memory size estimation could be present in that trade-off, however the importance of the estimation is not very significant. Since the pre-GLT and GLT trade-offs will be addressed in separate subsections due to the importance of memory size estimation in those trade-offs, we are going to briefly discuss below the two remaining trade-offs displayed at the bottom of the flow diagram (Fig. 7).

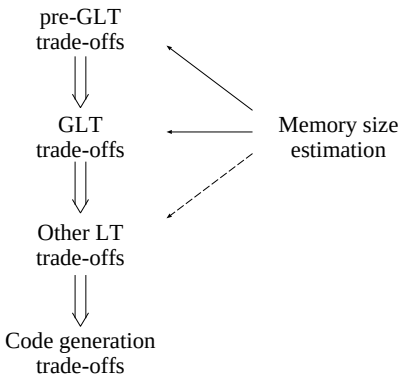


Figure 7: Memory size estimation as part of the trade-off oriented loop transformation (LT) flow chart (GLT stands for *Global Loop Transformations*).

Trade-offs w.r.t. the data memory size appear, especially, when we enlarge the exploration space or change the execution ordering. After GLT, the global execution ordering is fixed and, hence, less opportunities remain. In particular, in the subsequent steps of the DTSE methodology also other loop transformations (LT) are applied to improve, e.g., the performance and bandwidth related aspects [49], [50]. These loop transformations do not have such a big impact on the data memory size as the GLT. Due to that, the high-level memory estimators are not needed that badly in this phase and, hence, are not used so often. Still, they can he helpful also in this phase as suggested by the dashed arrow in Fig. 7. Also at the end during the code generation from the geometrical model, interesting trade-offs exist. During this phase when also the local execution ordering is fixed, we trade-off the code size, the complexity of the control-flow, and the number of empty iterations for the same geometrical model representation from which different codes can be generated. These trade-offs have already been discussed by the authors working on code generators from geometrical model [51], [52]. Moreover, they do not affect the data memory size, so we shall not discuss this phase 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 [53], [54], [55], [56], [57], [58]. However, the model imposes strict limitations on the input code. It can deal only with static control parts [59]. 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 [2]. 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 [48], pointer analysis and conversion [60], [61], dynamic single assignment conversion [62], advanced copy propagation [63], hierarchical rewriting [2], and scenario creation [64], [65], [66], 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 [67] to the GLT trade-offs which will be discussed in the next subsection, i.e., the constraints created during the decisions in the pre-GLT trade-offs are propagated and constrain the GLT trade-offs.

4.2 GLT trade-offs

After preparing the application to be parsed for the geometrical model analysis, 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 elements that are not needed any more for the future production of structures or elements [45]. The closer the production and the consumption of the data structure or data structure elements in the program, the better the in-place optimization step can be applied. Data reuse optimization aims at placing a local copy of the part of the array which will be used (consumed) several times, closer to the data path [4]. The closer the consumptions in the program, the better the data reuse will be. However, the close consumption for improved 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 in trade-offs between the number of memory accesses and the data memory size. An example of this trade-off, combined 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 due to the irregular access. However, high-efficient code should not contain too complex control flow since it will slow down the application on the processor. Especially on VLIW processors this can cause much overhead because, due to the conditions, the loops can have a problem with the crucial software pipelining substep. The rich control flow in the application can create overhead if good branch prediction or guarded execution is missing in the processor data path hardware. The extra control flow is mainly present because of fusion and shifting of the loops to obtain optimal locality and to still satisfy the flow dependencies. Fortunately, this control flow overhead can be reduced by not fusing all the loops that are required for optimal locality. This will cause some growth of the memory size requirements, resulting in a data memory size vs. control flow complexity trade-off. This trade-off can be combined with the trade-off between intra in-place and data reuse as we will show in Section 4.3 for a real-life application. Note, that the memory size evaluation techniques from Section 3 are crucial when steering or providing early feedback to the designer w.r.t. those complex memory oriented trade-offs.

4.3 The GLT trade-off educative demonstrator

During all trade-offs discussed in the previous subsections the memory size estimation is beneficial. We shall demonstrate it first on the simple example of trade-off between storage size and number of memory accesses in *Example 4*. Later, in Section 5.2, we provide also results for a real-life test-vehicle.

In the simple example, three 4x5 arrays are present: *A*, *B* and *C*, embedded in 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 reversed loop iterator in the second loop nest. To reverse the execution of 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, a 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 requires 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 interesting option is the loop fusion of the last two loop nests. This decreases 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 [68] 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 Experimental Results

This section provides quantitative results, allowing to better understand the complementarity of the techniques discussed in this paper. First, we provide the experimental results for memory size estimation and optimization for a number of real-life benchmarks. This is followed by a case study of the different trade-offs during GLT which have been discussed in Section 4.

5.1 Experimental results for memory size estimation and optimization

In this subsection, we present several experiments with three software tools:

(1) K2 is a tool computing exactly the minimum data memory size, developed at the University of Illinois at Chicago, whose model was presented in Section 3.2;

(2) STOREQ is a memory size estimation tool, developed at the Norwegian University of Science and Technology, Trondheim, Norway, whose model was discussed in Section 3.1;

(3) MC is a Memory Compaction tool from the DTSE design flow (see Section 4), developed at the Interuniversity Microelectronics Center (IMEC), Leuven, Belgium.

MC performs low-level storage order optimization of multi-dimensional data [45], which is normally the last step in the DTSE design flow. This hence gives examples of reasonable implementations, using advanced techniques that perform a trade-off between memory size and address and code complexity.

In order to make possible the comparative evaluation of these techniques, application codes were considered *procedural*, therefore the execution ordering was entirely fixed, as implied by the code organization. The selected benchmarks are either algebraic kernels or applications from digital signal processing: (1) a real-time regularity detection algorithm used in robot vision; (2) Durbin's algorithm for solving Toeplitz systems with N unknowns; (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 motion detection algorithm used in the transmission of real-time video signals on data networks [1]; (5) the kernel of a motion estimation algorithm for moving objects (MPEG-4).

Table 3 summarizes the results of our experiments. The STOREQ and MC tools have been run on a Linux server with a Dual 2.4 GHz Xenon Processor and 3 GB RAM. K2 was run on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory.[2] Columns "# Array Refs." and "# Scalars" display the numbers of array references and, respectively, scalar signals (array elements) in the application codes. For each of the three tools, column "Memory" displays the

[2]The difference between the two testing platforms does not affect in a significant way the relative differences between running times. According to [71], the former platform may be slightly faster (no more than 10-15%) than the latter, especially for applications exploiting the presence of the dual processor. The less amount of RAM of the second platform does not affect the performance of the K2 tool since this is not a critical resource.

Application (parameters)	# Array Refs.	# Scalars	K2		STOREQ		MC	
			Memory	CPU	Memory	CPU	Memory	CPU
Regularity detection (MaxGrid=8, L=64)	19	4,752	2,304	0.8	4,153	2.2	3,706	1.3
Durbin algorithm (N=500)	27	252,499	1,249	15	2,002	2.5	1,502	40.4
2-D Gauss blue filter (N=800,M=600)	20	5,260,027	480,005	137	958,809	2.0	958,805	2.2
Motion detection (M=N=32, m=n=4)	11	72,543	2,740	2	2,770	1.6	2,741	1.4
(M=N=120, m=n=8)		3,749,063	33,284	16	33,398	1.8	33,285	7.9
MPEG-4 motion estimation	68	265,633	2,465	18	3,399	2.0	3,396	1.6

Table 3: Experimental results

storage requirements obtained (numbers of memory locations) and column "CPU" shows the corresponding running times in seconds.

Table 3 shows that both STOREQ and MC are usually much faster than the exact computation approach (although the running times of the latter are still reasonable). This is not unexpected: performing exact computations typically implies a higher computational effort. Sometimes, the amount of additional computations can prove to be very significant, as in the case of the 2-D Gaussian blur filter application. This happens mainly when the data storage has relative small variations during the code execution, preventing the pruning mechanism of the tool to work efficiently. As exemplified in [37], the tool K2 detects and eliminates from further analysis the blocks of code where the local maxima cannot exceed the overall storage requirement. This "pruning" speeds up the tool, concentrating the analysis on those portions of code where the memory increase is likely to happen. Consequently, the tool is slower for applications having very many local maxima of storage variation, as it happens during the execution of the 2-D Gaussian blur filter application (see the left part of the memory traces in Fig. 5). On the other hand, the more computationally-expensive exact technique can be used to obtain the exact minimum data storage (about half of the values found by the estimation tools) and precise memory traces, like the ones in Fig. 5, allowing to identify the parts of the application code requiring more storage.

The minimum storage requirements are difficult to use in practical allocation problems (typically requiring significantly more complex hardware for address generation). MC performs a trade-off between memory size and the complexity of the address generation hardware. This causes, for instance, the big difference between the exact size of the 2-D Gaussian blur filter found by K2 and the result of MC (and estimate of STOREQ) for this application. On the other hand, knowing the minimum storage gives a good idea on this compromise in terms of extra storage.

As it can be seen from Table 3, the results of STOREQ are, in general, quite close to the implementations obtained using MC. The goal of STOREQ is not, like K2, to find the smallest possible memory requirement, but to estimate the size of the memory required by a likely implementation. This is also useful at the early loop-transformation stage. The optimized implementation found by MC is assumed to be such a likely implementation. The overestimate of STOREQ compared to MC in the Durbin algorithm is caused by a bounding box around array elements produced along a two-dimensional diagonal line in iteration space. If this is unacceptable, a more detailed dependency analysis must be performed, so that these cases can be treated separately. These situations appear quite rarely in our application domain. STOREQ is very fast, enabling use of the tool during the early design step of loop transformation exploration. Even

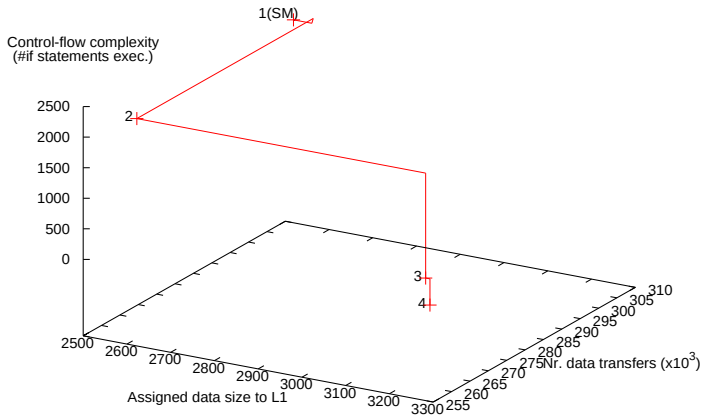


Figure 8: The 3-D exploration space example (#mem. accesses×data storage size×control flow complexity) for a real life multimedia application (QSDPCM [69]) [70].

more so since most of the time reported here (over 80% on the average) stems from the first data dependency analysis step described in Section 3.1. This part does not have to be repeated if the sizes of multiple alternative (partially fixed) execution orderings are estimated. It will then only be the CPU time for size estimation, 0.3 seconds on average for the applications in Table 3, that will be required. Notice also that the STOREQ run-time is close to independent of the number of nodes in a given iteration space, as demonstrated by the two versions of the motion detection kernel. Furthermore, the tool is implemented in Matlab. When a future version is available in C++, like the other tools, it will be substantially faster. Together this will make the STOREQ approach useful even for automatic loop transformation exploration where the search space can have thousands of possible solutions. For this it will not be possible to use MC or K2.

5.2 The GLT trade-off real-life demonstrator

Subsection 4.3 demonstrates an educative example which trades-off the data memory size vs. the number of memory accesses. This subsection demonstrates a more complex, three dimensional trade-off among the data memory size, the number of memory accesses, and the control-flow complexity for the real-life QSDPCM demonstrator [69]. This trade-off – depicted in Fig. 8 – is the combination of the trade-offs which have been discussed in Subsection 4.2 and has been first presented in more detail in [70]. The different points in Fig. 8 represent code versions with different loop transformations applied. These transformations are strip mining, loop fusion, and shifting.

In this paper, we only analyze this trade-off w.r.t. the use of the high-level storage size estimators presented in Section 3. To analyze the storage size usage, we use two approaches: the high-level STOREQ estimator approach, based on the size of the data flow dependency, and the exact memory trace (which is part of the K2 tool) for the array that participates in the data flow dependency. First we discuss the high-level estimation approach and then we show on the memory traces that the estimation is in-line with the exact memory trace provided by the K2 tool.

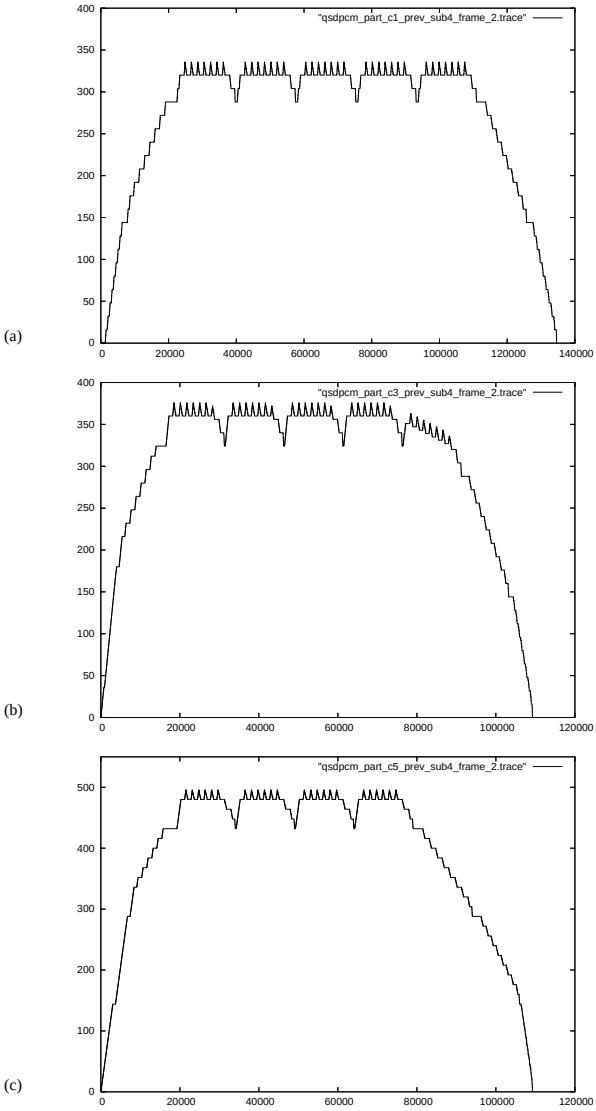


Figure 9: Memory traces for several QSDPCM versions of code (1,2, and 4) with different storage size requirements.

We start our exploration from the code version that has the best memory size due to the high-level memory estimators. This point is named 1(SM) in our exploration space. In the next code version (going from point 1(SM) to 2), we have observed that the data storage size increases from 2,544 elements to 2,589 elements. This has two effects. First, we have decreased the data transfers from 307×10^3 to 259×10^3. Second, the number of *if* statement evaluations has increased from 2,206 to 2,288. When going from point 2 to 3, we have reduced the number of *if* statement evaluations from 2,288 to 572, i.e., by a factor of 4, for an increase in memory size from 2,589 to 3,249. In the last code version (going from point 3 to 4), we have decreased the number of *if* condition evaluations even further, again by a factor of 4. This improvement is obtained with a minor cost – the increase of memory size by only 9 locations. In this demonstrator, we can see again that the exploitation of high-level storage size estimation together with other high-level estimations is crucial when positioning a code version in the exploration space and determining the Pareto optimality of the code version.

We validate the memory size estimation with an exact memory trace (obtained with the K2 tool whose model was presented in Section 3.2). The memory traces are depicted in Fig. 9. As already explained in Section 3.2.2, the abscissae are the numbers of datapath instructions in the code and the ordinates are the memory locations in use. Figure 9(a) shows the memory trace for the code version 1(SM), Figure 9(b) shows the memory trace for the code version 2, and Figure 9(c) shows the memory trace for the code version 4. As mentioned above, these versions have the same functionality and differ only by the loop transformations applied. Note that this memory trace is only for one affected array. We can observe the same relative behavior of the peak values of the memory size as in Fig. 8. That is, the relative increase of the estimation and the peak value of the memory trace do match. The best code w.r.t. the memory size is version 1(SM) with 336 storage locations, the next one is version 2 with 376 locations, and the worst is version 4 with 496 locations for one affected array in the application. However, as stated above, this last version has other benefits resulting from the discussed trade-offs like less data transfers and less *if* statement evaluations (see also Fig. 8).

6 Conclusions

This paper has given an overview on the techniques for the evaluation of the data memory size in signal processing applications, focusing on the most advanced developments in this research domain. Two alternative techniques, an estimation approach – investigating the different loop execution orders in non-procedural specifications, and a computation approach – finding the exact minimum data memory size – have been presented. The 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] 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*, Kluwer Academic Publishers, Boston, 1998.

[2] P.R. Panda, F. Catthoor, N. Dutt, K. Dankaert, E. Brockmeyer, C. Kulkarni, P.G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Trans. Design Automation of Electronic Syst.*, vol. 6, no. 2, pp. 149-206, April 2001.

[3] 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.

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

[5] 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," in *Proc. Asia & S.-Pacific Design Automation Conf.*, Yokohama, Japan, Jan. 2006, pp. 606-611.

[6] I.I. Luican, H. Zhu, and F. Balasa, "Formal model of data reuse analysis for hierarchical memory organizations," in *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, San Jose CA, Nov. 2006, pp. 595-600.

[7] E. Brockmeyer, M. Miranda, F. Catthoor, H. Corporaal, "Layer assignment techniques for low energy in multi-layered memory organisations," in *Proc. ACM/IEEE Design Automation and Test in Europe Conf.*, Munich, Germany, March 2003, pp. 1070-1075.

[8] C.J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. on Comp.-Aided Design of ICs and Syst.*, vol. CAD-5, no. 3, pp. 379-395, July 1986.

[9] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man, "An efficient microcode compiler for custom DSP processors," in *Proc. IEEE Int. Conf. Comp.-Aided Design*, Santa Clara CA, Nov. 1987, pp. 24-27.

[10] F.J. Kurdahi and A.C. Parker, "REAL: A program for register allocation," in *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 210-215.

[11] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proc. 8th Design Automation Workshop*, 1971, pp. 155-169.

[12] G. Goossens, *Optimization Techniques for Automated Synthesis of Application-specific Signal-processing Architectures*, Ph.D. thesis, K.U. Leuven, Belgium, 1989.

[13] P.G. Paulin and J.P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. on Comp.-Aided Design of ICs and Syst.*, vol. 8, no. 6, pp. 661-679, June 1989.

[14] C.H. Gebotys and M.I. Elmasry, *Optimal VLSI Architectural Synthesis*, Boston: Kluwer Academic Publ., 1992.

[15] L. Stok and J. Jess, "Foreground memory management in data path synthesis," *Int. J. Circuit Theory and Appl.*, vol. 20, pp. 235-255, 1992.

[16] K.K. Parhi, "Calculation of minimum number of registers in arbitrary life time chart," *IEEE Trans. Circ. & Syst. - II: Analog and Digital Signal Processing*, vol. 41, no. 6, pp. 434-436, 1994.

[17] S.Y. Ohm, F.J. Kurdahi, and N. Dutt, "Comprehensive lower bound estimation from behavioral descriptions," in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, 1994, pp. 182-187.

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

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

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

[21] P. Grun, F. Balasa, and N. Dutt, "Memory size estimation for multimedia applications," in *Proc. 6th Int. Workshop Hardware/Software Co-Design*, Seattle WA, March 1998, pp. 145-149.

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

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

[24] F. Balasa, F. Catthoor, and 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.

[25] F. Balasa, F. Catthoor, and 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.

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

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

[28] IMEC, *Atomium web site* [Online]. Available: http://www.imec.be/design/atomium/ .

[29] Q. Hu, A. Vandecappelle, P. G. Kjeldsberg, F. Catthoor, and M. Palkovic, "Fast Memory Footprint Estimation based on Dependency Distance Vector Calculation," in *Proc. ACM/IEEE Design Automation and Test in Europe*, Nice, France, April 2007, pp. 379-384.

[30] P.G. Kjeldsberg, F. Catthoor, and 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.

[31] 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.

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

[33] M. Moonen, P. V. Dooren, and J. Vandewalle, "An SVD updating algorithm for subspace tracking," SIAM J. Matrix Anal. Appl., vol. 13, no. 4, pp. 1015-1038, 1992.

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

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

[36] H. Zhu, I.I. Luican, and F. Balasa, "Memory size computation for multimedia processing applications," in *Proc. Asia & South-Pacific Design Automation Conf.*, Yokohama, Japan, Jan. 2006, pp. 802-807.

[37] 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.

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

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

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

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

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

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

[44] 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.

[45] 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 Multi-media," A. Krikelis (ed.), in *Parallel Computing*, Elsevier, vol. 23, no. 12, Dec. 1997.

[46] 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.), 2002, pp. 167-181.

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

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

[49] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis, "Power, Performance and Area Exploration for Data Memory Assignment of Multimedia Applications," in *Proc. Systems, Architectures, Modeling, and Simulation*, A. Pimentel and S. Vassiliadis (eds.), LCNS, vol. 3133, Springer-Verlag, Samos, Greece, June 2004, pp. 540-549.

[50] K.C. Shashidar, A. Vandecappelle, and F. Catthoor, "Low power design of turbo decoder module with exploration of energy-performance trade-offs," in *Proc. Workshop on Compilers and Operating Systems for Low Power* in conjunction with *Int. Conf. on Parallel Arch. and Compilation Techniques*, Barcelona, Spain, Sept. 2001, pp. 10.1-10.6.

[51] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. Int. Conf. on Parallel Arch. and Compilation Techniques*, Juan-les-Pins, France, Sept. 2004, pp. 7-16.

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

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

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

[55] A. Darte and 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.

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

[57] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," Univ. Maryland College Park, CS-TR-3193, Apr. 1993.

[58] M.E. Wolf and M.S. Lam, "A data locality optimizing algorithm," in *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 30-43.

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

[60] L. Semeria and G. De Micheli, "SpC: synthesis of pointers in C," in *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, Santa Clara CA, Nov. 1998, pp. 340-346.

[61] B. Franke and 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.

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

[63] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, "Advanced copy propagation for arrays," in *Proc. SIGPLAN Conf. Languages, Compilers, and Tools for Embedded Syst.*, San Diego CA, June 2003, pp. 24-33.

[64] M. Palkovic, H. Corporaal, and F. Catthoor, "Global memory optimisation for embedded systems allowed by code duplication," in *Proc. 9th Int. Wsh. on Software and Compilers for Embedded Systems*, Sept. 2005, pp. 72-80.

[65] M. Palkovic, H. Corporaal, and F. Catthoor, "Dealing with data dependent conditions to enable general global source code transformations," *Int. J. of Embedded Systems*, Interscience Publ., in press, 2007.

[66] M. Palkovic, E. Brockmeyer, P. Vanbroekhoven, H. Corporaal, and F. Catthoor, "Systematic preprocessing of data dependent constructs for embedded systems," *J. Low Power Electronics*, American Scientific Publ., vol. 2, no. 1, pp. 9-17, April 2006.

[67] F. Catthoor and E. Brockmeyer, "Unified meta-flow summary for low-power data-dominated applications," chapter in *Unified low-power design flow for data-dominated multi-media and telecom applications*, F. Catthoor (ed.), Boston: Kluwer Acad. Publ., 2000, pp. 7-23.

[68] V. Pareto, *Cours D'Economie Politique*, volume I–II, Lausanne, 1896.

[69] P. Strobach, "QSDPCM – A new technique in scene adaptive coding," in *Proc. 4th Eur. Signal Processing Conf.*, Grenoble, France, Amsterdam: Elsevier Publ., Sept. 1988, pp. 1141-1144.

[70] M. Palkovic, H. Corporaal, and F. Catthoor, "Trade-offs in loop transformations," submitted to *ACM Trans. on Design Automation of Electronic Systems*, 2007.

[71] * * * , *Benchmark Marathon: 65 CPUs from 100 MHz to 3066 MHz* [Online]. Available: http://www.tomshardware.com/2003/02/17/benchmark_marathon/index.html