

Storage Requirement Estimation for Optimized Design of Data Intensive Applications

P. G. KJELDSBERG

Norwegian University of Science and Technology

and

F. CATTLOOR

Inter-University Microelectronics Center and Katholieke Universiteit Leuven

and

E. J. AAS

Norwegian University of Science and Technology

A novel storage requirement estimation methodology is presented for use in the early system design phases when the data transfer ordering is only partially fixed. At that stage, none of the existing estimation tools are adequate, as they either assume a fully specified execution order or ignore it completely. A prototype CAD tool has been developed that includes major parts of the storage requirement estimation and optimization methodology. Using representative application demonstrators, we show how our techniques and tool can effectively guide the designer to achieve a transformed specification with low storage requirement.

Categories and Subject Descriptors: B.3 [**Hardware**]: Memory Structures; B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Memory Design*; B.5.1 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; *Optimization*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: size estimation, data optimization, code transformation, memory architecture exploration, high-level synthesis

1. INTRODUCTION

Many integrated circuit systems, particularly in the multi-media and telecom domains, are inherently data dominant. For this class of applications, data transfer and storage largely determine cost and performance parameters. This is the case for chip size, since large memories are usually needed, performance, since access-

Author's address: P. G. Kjeldsberg, Norwegian University of Science and Technology, Trondheim, Norway; email: pgk@fysel.ntnu.no; F. Catthoor, Inter-University Microelectronics Center and Katholieke Universiteit Leuven, Leuven, Belgium; email: catthoor@imec.be; E. J. Aas, Norwegian University of Science and Technology, Trondheim, Norway; email: aas@fysel.ntnu.no.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 1084-4309/20YY/0400-0001 \$5.00

ACM Transactions on Design Automation of Electronic Systems, Vol. V, No. N, MM 20YY, Pages 1–0??.

This is the author's version of the work. It is posted here by permission of ACM for your personal use.
Not for redistribution. The definitive version will be published by ACM in the near future. See copyright notice above.

ing the memories may very well be the main bottleneck, and power consumption, since the memories and buses consume large quantities of energy. Even for systems with caches, the overall storage requirement has vital impact on the performance and power consumption, since it greatly influences the number of slow and power expensive cache misses. For the system development process, the designer must hence concentrate first on exploring the data transfer and storage to achieve a cost optimized end product [Catthoor et al. 1998].

At the system level, no detailed information is available about the size of the memories required for storing data in the alternative realizations of the application. To guide the designer and assist in choosing the best solution, estimation techniques for the storage requirements are needed, very early in the system design trajectory. For our target classes of data dominant applications the high level description is typically characterized by large multi-dimensional loop nests and arrays. A straightforward way of estimating the storage requirement is for each array to multiply the size of its dimensions, and then add together the sizes of the different arrays. This will normally result in a huge overestimate however, since not all the arrays, and possibly not all parts of one array, are alive at the same time. In this context an array element is alive from the moment it is written, or produced, and until it is read for the last time. This last read is said to consume the element. To achieve a more accurate estimate, we have to take into account these non-overlapping lifetimes and their resulting opportunity for mapping arrays and parts of arrays in the same place in memory, the so-called in-place mapping problem. To what degree it is possible to perform in-place mapping depends heavily on the order in which the elements in the arrays are produced and consumed. This is mainly determined by the execution ordering of the loop nests surrounding the statements accessing the arrays.

At the beginning of the system design process, no information about the execution order is known, except what is given from the data dependencies between the statements in the code. As the process progresses, the designer makes decisions that gradually fix the ordering, until the full execution ordering is known. No exact memory size can be determined at these stages as long as the execution order is not fixed. So to steer this process, the best strategy to aid the system designer at each transformation step appears to come from estimates of the upper and lower bounds on the storage requirement, given the partially fixed execution ordering. Even though such bounds are far from trivial to derive, we will show in this paper that this is both feasible and efficiently achievable within realistic assumptions on the application code.

After a discussion of previous work in the field, the main ideas behind a novel storage requirement estimation technique is presented. We then introduce a prototype CAD tool employing these techniques, before we show how it can be used during design of real life applications.

2. PREVIOUS WORK

2.1 Scalar-Based Estimation

By far the major part of all previous work on storage requirement has been scalar based. 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, the lifetime of each scalar is found so that scalars with non-overlapping lifetimes can be mapped to the same storage unit [Kurdahi and Parker 1987]. Techniques such as clique partitioning are also exploited to group variables that can be mapped together [Tseng and Siewiorek 1986]. In [Ohm et al. 1994], a lower bound for the register count is found without the fixation of a schedule, through the use of As-Soon-As-Possible and As-Late-As-Possible constraints on the operations. A lower bound on the register cost can also be found at any stage of the scheduling process using Force-Directed scheduling [Paulin and Knight 1989]. Integer Linear Programming techniques are used in [Gebotys and Elmasry 1991] to find the optimal number of memory locations during a simultaneous scheduling and allocation of functional units, registers and busses. A thorough introduction to the scalar-based storage unit estimation can be found in [Gajski et al. 1994]. 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.

2.2 Estimation Techniques for Multi-Dimensional Arrays

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 code's loop structure, 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.

In [Verbauwhede et al. 1994], 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. To be able to generate the production time axis and production and consumption dates, the execution ordering has to be fully fixed. Also, since each array is treated separately, only in-place mapping internally to an array (intra-array in-place) is considered, not the possibility of mapping arrays in-place of each other (inter-array in-place).

Another approach is taken in [Grun et al. 1998]. Assuming procedural execution of the code, 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.

[Zhao and Malik 1999] 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, as well as in our methodology, 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 [Ramanujam et al. 2001], 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. The technique assumes a fully fixed execution ordering. 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.

In contrast to the methods described so far in this subsection, the storage requirement estimation technique presented in [Balasa et al. 1995] does not take execution ordering into account at all. It starts with an extended data dependency analysis resulting in a number of non-overlapping basic sets and the dependencies between them. The basic sets and dependencies are described as polytopes, using *linearly bounded lattices* (LBLs) of the form

$$x = T \bullet i + u | A \bullet i \geq b$$

where $x \in Z^m$ is the coordinate vector of an m -dimensional array, and $i \in Z^n$ is the vector of n loop iterators. The array index function is characterized by $T \in Z^{m \times n}$ and $u \in Z^m$, while the polytope defining the set of iterator vectors is characterized by $A \in Z^{2n \times n}$ and $b \in Z^{2n}$. The basic set sizes, and the sizes of the dependencies, are found using an efficient lattice point counting technique. The dependency size is the number of elements from one basic set that is read while producing the depending basic set. The total storage requirement is found through a greedy traversal of the corresponding data flow graph. The maximal combined size of simultaneously alive basic sets gives the storage requirement. Since no execution ordering is taken into account, all elements of a basic set are assumed produced before the first element is consumed. This gives rise to an overestimate compared to all but the worst-case ordering.

In summary, all of the previous work on storage requirement entails a fully fixed execution ordering to be determined prior to the estimation. The only exception is the last methodology, which allows any ordering not prohibited by data dependencies. None of the approaches permit the designer to specify partial ordering constraints, which is really essential during the early exploration of the system

level code transformation. In the next section we will present our new estimation technique which allows a partially fixed execution ordering.

As will be demonstrated in Section 5, the main usage of storage requirement estimation, is for optimization of an application's overall memory size. Examples of work in the field of memory size optimization through use of in-place mapping include [Quillere and Rajopadhye 2000] [Lefebvre and Feautrier 1997] [De Greef et al. 1997]. All of them require a fully fixed execution ordering. Much research has also gone into estimation and optimization of size, performance and power consumption of memory hierarchies in cache based systems. Examples of this are [Chakrabarti 2001] [Kirovski et al. 1999] [Panda et al. 1999]. The parallel compiler community also has much related work [Wolfe 1996]. Our memory size estimates can be used by these compilers as additional guidance while comparing alternative approaches for performance optimization. Further details are outside the scope of this paper.

3. ESTIMATION WITH PARTIALLY FIXED EXECUTION ORDERING

3.1 Motivation and Context

The new estimation methodology employs the data flow graph generation presented in [Balasa et al. 1995]. In this paper we mainly focus on data dependency size estimation however, which can be utilized for most polyhedral dependency descriptions. The main improvement compared to previous methods is the possibility to avoid overestimates by taking whatever execution ordering information available into account. Using an estimation methodology that requires a fully fixed ordering necessitates a full exploration of all alternatives for the unfixed parts. The complexity of investigating for example all loop interchange solutions for a loop nest without any constraints is $N!$, where N is the number of dimensions in the loop nest. For realistic examples, N can be as large as six even if the dimensionalities of the individual arrays are smaller. Since it is the number of loop dimensions that determine the complexity, a full exploration is very time consuming and hence not feasible when the designer needs fast feedback regarding an application's storage requirement when the execution ordering is not fully fixed. Instead, our methodology presents precise upper and lower bounds to the designer as guidance during the early high level design trajectory. Throughout the application code there may be conflicting dependency considerations, so an automatic tool is needed to lead the designer to a globally efficient solution.

Our algorithm is useful for a large class of applications. Certain restrictions exist on the code that can be handled in the present version however, some of which will be alleviated through future work. The main requirements are that the code is single assignment and has affine array indexes. This is achievable by a good array data flow analysis preprocessing, see [Feautrier 1991] and [Pugh and Wonnacott 1993]. Also the resulting Dependency Parts, see below, must be orthogonal, or made orthogonal, as described in [Kjeldsberg et al. 2000b]. When in the sequel the words upper and lower bounds are used, we mean the bounds after orthogonalization. The lower bounds may not be factual, since some approximations tend to result in overestimates so that realizations with lower storage requirements may exist. For a large class of applications, where the Dependency Parts are already orthogonal,

```

for i=0 to 5 {
  for j=0 to 5 {
    for k=0 to 2 {
      S.1  A[i][j][k] = f1( input );
      S.2  if(i>=1)&(j>=2) B[i][j][k] = f2( A[i-1][j-2][k] );
    }}

```

Fig. 1. Code example for concept definitions.

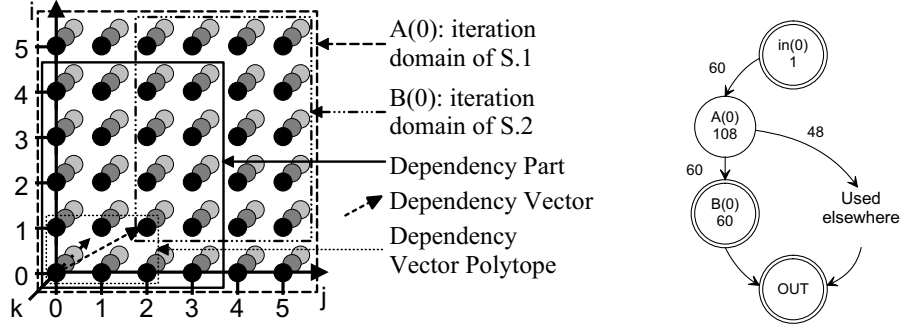


Fig. 2. Iteration space and data-flow graph of example in Figure 1.

and when the generation of the best-case and worst-case common iteration space is possible, both bounds are correct.

Let us take a look at the simple application code example shown in Figure 1. Two statements, S.1 and S.2, 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 the statements [Banerjee 1988].

The loops around the statements define an *iteration space*, [Banerjee 1988], as shown in Figure 2. Each node within this space represents one execution of the statements inside the loop nest. For our example, at each of these *iteration nodes* one A -array element and, when the if clause condition is true, one B -array element is produced. These nodes also constitute the *iteration domains* of the two statements, as indicated by the rectangles $A(0)$ and $B(0)$ in Figure 2. To avoid unnecessarily complex figures, two-dimensional rectangles are used. All nodes within the two rectangles are part of the corresponding iteration domains. Figure 2 also shows the corresponding data-flow graph for the code example in Figure 1. The nodes are the iteration domains, while the dependencies between them are the branches.

In general not all elements produced by one statement are read by a depending statement. A *Dependency Part* (DP) is therefore defined containing the iteration nodes for 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 producing the depending element. This DV spans a *Dependency Vector Polytope* (DVP) and its dimensions are defined as *Spanning Dimensions* (SD). Since the SD normally only comprises a subset of the iterator space dimensions, the remaining dimensions are denoted *Nonspanning Dimensions* (ND). The largest value for each dimension in the DVP is denoted

$$\begin{aligned}
DP_{A(0)-B(0)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -4 \\ 0 \\ -3 \\ 0 \\ -2 \end{bmatrix} \right. \\
DVP_{A(0)-B(0)} \begin{bmatrix} u \\ v \\ w \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \left| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1 \\ 0 \\ -2 \\ 0 \\ 0 \end{bmatrix} \right.
\end{aligned}$$

Fig. 3. LBL description of DP and DVP for A(0) with respect to B(0) of example in Figure 1.

Spanning Value (SV) of that dimension. For the DVP in Figure 2, i and j are SDs while k is ND. Furthermore we have $SV_i = 1$ and $SV_j = 2$.

The DP and DVP are important concepts, as they are used to estimate the upper and lower bounds on the size of a dependency respectively. The following definition is used for the dependency size.

DEFINITION 3.1. *The size of a dependency between two iteration domains 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. The DVP will hold the iteration nodes that certainly will be visited before the first depending iteration node, while the DP will hold the iteration nodes that in the worst case may be visited. This is discussed in detail in Section 3.4.

The DP and DVP can be represented using an LBL description as shown in Figure 3. The vertical line in the LBLs divides the description into an array index function and a restriction part. The restricted function for the u index of $DP_{A(0)-B(0)}$ can thus be read as $u = i$ for $0 \leq i \leq 4$. Note that for comparison with the DP a three-dimensional LBL is used for the DVP, even though it only has two dimensions, i and j .

In many cases, the end point of the DV falls outside the DP. This happens when no overlap exists between the two depending iteration domains. The DVP is then intersected with the DP.

When a dependency is not uniform, the DVs for different iteration nodes within its DP can differ in both length and direction. The DVP is then generated using the extreme DVs as introduced in [Danckaert 2001]. Figure 4 demonstrates the concept using a small illustrative code example. The different DVs for each individual iteration node are indicated in the left part of the Figure, while the resulting extreme DV is shown to the right. Section 5.1.1 gives an example of a more complex DVP generation. Use of the extreme DVs is a simple approach to uniformization suitable for estimation when a short run time is required to give fast feedback to the designer. The final implementation of an algorithm will often require a linear storage

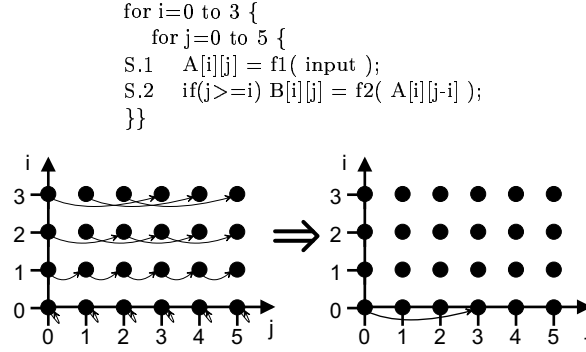


Fig. 4. Uniformization using extreme DV.

order to avoid overly complex address generation. As for the overestimates induced by orthogonalization, the possible overestimates resulting from our uniformization will then actually give the correct storage requirement. Alternative techniques for uniformization of affine dependency algorithms are presented in [Shang et al. 1996]. The complexity of these procedures is however much larger.

With the dependency size estimation methodology from [Balasa et al. 1995], the size of the dependency between S.1 and S.2 in Figure 1 will be the number of iteration nodes in the full DP, that is 60. This is an overestimate even for the worst-case ordering since the basic sets are overlapping. Inside this overlap both *A*-array elements and *B*-array elements are produced, and the *B*-array elements produced by S.1 can thus be mapped in-place of the *A*-array elements consumed by S.2. This overlap should therefore be removed, reducing the dependency size estimate to 36. An even more accurate estimate can be achieved by also taking available partially fixed execution ordering into account. The new estimation methodology presented in this paper is able to do this, producing upper and lower bounds on the dependency sizes and the overall storage requirement of an application. This will be discussed in the next section.

3.2 Overall Estimation Strategy

The storage requirement estimation methodology can be divided into four steps as shown in Figure 5. The first step uses the technique from [Balasa et al. 1995] to generate a data-flow graph reflecting the data dependencies in the application code. Any complete polyhedral dependency model can be used however, and work is in progress to integrate the estimation methodology with the ATOMIUM tool, [IMEC 2003] [Bormans et al. 1999]. The second step places the DPs of the iteration domains in a common iteration space to enable the global view of the fourth step. Best-case and worst-case placements may be used to achieve lower and upper bounds respectively. [Danckaert et al. 2000a; 2000b] describes work that can be utilized for this step. The third step focuses on the size of individual dependencies between the iteration domains in the common iteration space. The final step searches for simultaneously alive dependencies in the common iteration space, and calculates their maximal combined size.

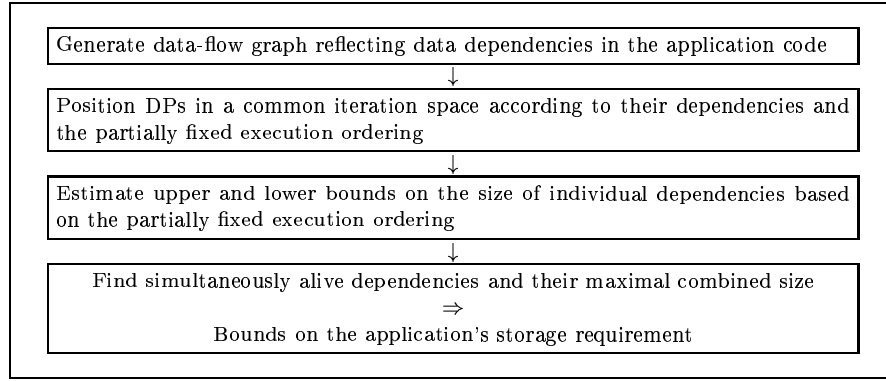


Fig. 5. Overall storage requirement estimation strategy.

The remainder of this Section will first give a short description of a simplified technique for generation of a common iteration space. This is followed by a detailed introduction to the principles of the third step. Finally, an overview of the last step is presented. The main objective is to give the reader adequate understanding of the techniques to follow the presentation of the prototype CAD tool in Section 4. A more accurate methodology for the third step can be found in [Kjeldsberg et al. 2000a], with further details in [Kjeldsberg et al. 2003]. More information regarding detection of simultaneously alive dependencies is presented in [Kjeldsberg et al. 2001].

3.3 Generation of Common Iteration Space

In the original application code, an algorithm is typically described as a set of imperfectly nested loops. At different design steps, parts of the execution ordering of these loops are fixed. The execution ordering may include both the sequence of separate loop nests, and the order and direction of the loops within nests. To perform global estimation of the storage requirement, taking into account the overall limitations and opportunities given by the partial execution ordering, a common iteration space for the code is needed. A common iteration space can be regarded as representing one loop nest surrounding the entire, or a given part of the code. This is similar to the global loop reorganization described in [Danckaert et al. 2000b]. Figure 6 shows a simple example of the steps required for generation of such a common iteration space. Note that even though it here includes rewriting the application code, this is not necessary to perform the estimation. The common iteration space can be generated directly using the PDG model of [Catthoor et al. 1998].

The introduction of the common iteration space opens for an aggressive in-place mapping which may not be used in the final implementation. The placement of the DPs in the common iteration space entails certain restrictions on the possible execution ordering of the still unfixed parts of the code. The sizes of the individual dependencies will hence be influenced by the placement of their DPs and depending DPs in the common iteration space. To have realistic upper and lower bounds it is therefore necessary to generate two common iteration spaces, one with a worst-

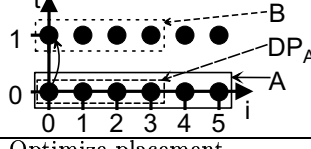
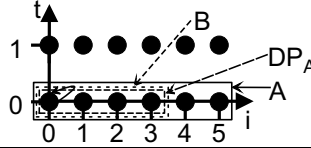
<pre> for (i=0; i<=5; i++) A[i] = ... for (j=0; j<=3; j++) B[j] = f(A[j]); </pre>	Original code
<pre> for (t=0; t<=1; t++) for (i=0; i<=5; i++) if (t==0) A[i] = ... for (t=0; t<=1; t++) for (i=0; i<=5; i++) if (t==1) & (i<=3) B[i] = f(A[i]); </pre>	Add outer pseudo-dimension Add/enlarge dimensions and add if-clauses to enable merging
<pre> for (t=0; t<=1; t++) for (i=0; i<=5; i++) { if (t==0) A[i] = ... if (t==1) & (i<=3) B[i] = f(A[i]); } </pre>	Merge loops 
<pre> for (t=0; t<=1; t++) for (i=0; i<=5; i++) { if (t==0) A[i] = ... if (t==0) & (i<=3) B[i] = f(A[i]); } </pre>	Optimize placement 

Fig. 6. Generation of Common Iteration Space.

case and one with a best-case placement, both taking into account the available execution ordering and other realistic design constraints. [Danckaert et al. 2000b] presents techniques for placement that enable the designer to organize the data accessing in such a way that aggressive in-place optimization can be employed. The worst-case placement can be based on a full loop body split, so that each statement is assigned individual iterator values in the pseudo t -dimension. For the rest of this paper, optimal iteration spaces are assumed, but the methodologies presented work equally well on alternative organizations of the iteration space.

3.4 Estimation of Individual Dependencies

Following Definition 3.1, the lower bound (LB) and upper bound (UB) on the storage requirement of a dependency can be found from the DVP and DP sizes respectively. As we will now see, these sizes may vary greatly with the chosen execution ordering. With no execution ordering fixed, the LB and UB can be estimated from the original DVP and DP. No matter what execution order is chosen in the end, the iteration nodes of the DVP will always be visited before the first depending iteration node. Similarly, no more than all iteration nodes, that is the full DP, can be visited before the first depending iteration node. If overlap exists between the DP/DVP and the depending iteration nodes, not all of their elements will be produced before the first depending element that can potentially be mapped in-place of the first DP/DVP element. The overlap can therefore be removed from

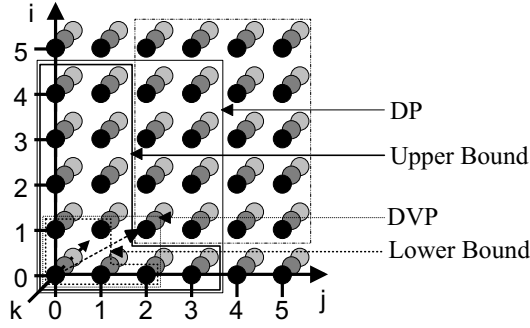


Fig. 7. DP and DVP and corresponding UB and LB for the dependency between $A(0)$ and $B(0)$ of example in Figure 1 without any execution ordering fixed.

Table I. Main principles for treatment of fixed dimensions.

	Fixation starts outermost	Fixation starts innermost
SD	Big expansion of DVP, small reduction of DP: Expand all unfixed SDs of the DVP to the boundary of the DP. Add all unfixed NDs to the DVP. <i>If</i> (no SD fixed so far): remove from the DP elements that for the current dimension have values larger than its SV. <i>Else</i> : remove elements that will not be visited from the DP	Small expansion of DVP, big reduction of DP: <i>If</i> (not last SD): expand the current dimension of the DVP to the boundary of the DP. Remove from the DP elements that for all unfixed SDs have values larger than their SV.
ND	Unchanged DVP, reduced DP: <i>If</i> (no SD fixed so far): remove the dimension from the DP <i>Else</i> : remove elements that will not be visited from the DP	Expanded DVP, unchanged DP: <i>If</i> (unfixed SDs still exist): add the dimension to the DVP

the DP and DVP. The DP and DVP in Figure 2 thus give the following estimates with no execution ordering fixed:

$$UB = \text{size}(DP) - \text{size}(DP_{\text{overlap}}) = 60 - 24 = 36$$

$$LB = \text{size}(DVP) - \text{size}(DVP_{\text{overlap}}) = 6 - 1 = 5$$

The UB and LB are shown graphical in Figure 7.

Unless no overlap exists between the DP and the depending iteration nodes, the bounds found using these simple principles cannot be reached. The detailed description of a more accurate estimation technique, but which is still based on the same principles, is given in [Kjeldsberg et al. 2003].

The execution ordering can be fixed through a fixation of dimensions starting with the innermost or outermost nest levels. SDs and NDs must be treated differently. Table I summarizes the estimation principles for each of these categories. The main effects on the DVP and DP are written in bold. If an ND is fixed outermost, only the DP changes, giving rise to a reduced upper bound and an unchanged lower bound. A fixation of an ND innermost has the opposite effect, an unchanged upper bound and an increased lower bound. If an SD is fixed outermost, both DVP and DP changes, giving rise to a reduced upper bound and an increased lower bound.

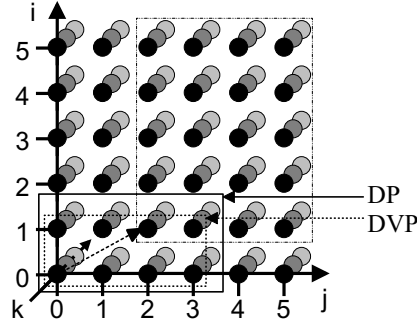


Fig. 8. DP and DVP of the dependency between $A(0)$ and $B(0)$ of example in Figure 1 with SD dimension j fixed innermost.

This is also the result if an SD is fixed innermost. In this last case however, the change in the DVP only covers the fixed SDs, while the fixation of an SD outermost effects all unfixed dimensions. The increase in the lower bound is therefore in most cases much smaller if the SD is fixed innermost. Note that there are many special cases that Table I does not explain fully, partly covered by the *Else* clauses. They are all handled by the more detailed methodology presented in [Kjeldsberg et al. 2003]. The methodology presented there also removes the requirement of fixation starting at the innermost or outermost nest level.

The main principles will now be demonstrated using the example code from Figure 1. Assume first that SD j has been fixed at the innermost nest level. The j dimension of the DVP is then extended to the boundary of the DP since these iteration nodes will now certainly be visited before the first depending iteration node. All iteration nodes of the DP with values in the i dimension larger then the SV_i (that is > 1) can be removed, since they are now certainly not visited before the first depending iteration node. The resulting DP and DVP are given in Figure 8. For enhanced readability the corresponding UB and LB are not drawn. As before, they are found through a removal of the overlap with the depending iteration nodes. The estimated bounds are now:

$$UB = \text{size}(\text{DP}) - \text{size}(\text{DP}_{\text{overlap}}) = 24 - 6 = 18$$

$$LB = \text{size}(\text{DVP}) - \text{size}(\text{DVP}_{\text{overlap}}) = 8 - 2 = 6$$

Alternatively, ND k can be fixed innermost. The k dimension is then added to the DVP, while the DP is unchanged. Figure 9 shows the DP and DVP for this partial ordering. The estimated bounds are now:

$$UB = \text{size}(\text{DP}) - \text{size}(\text{DP}_{\text{overlap}}) = 60 - 24 = 36$$

$$LB = \text{size}(\text{DVP}) - \text{size}(\text{DVP}_{\text{overlap}}) = 18 - 3 = 15$$

The nonspanning k dimension can instead be fixed at the outermost nest level. It is then removed from the DP, while the DVP remains unchanged as shown in Figure 10. The estimated bounds are now:

$$UB = \text{size}(\text{DP}) - \text{size}(\text{DP}_{\text{overlap}}) = 20 - 8 = 12$$

$$LB = \text{size}(\text{DVP}) - \text{size}(\text{DVP}_{\text{overlap}}) = 6 - 1 = 5$$

When one dimension is fixed either innermost or outermost the estimation can continue similarly for any additionally fixed dimension. Assume for instance that

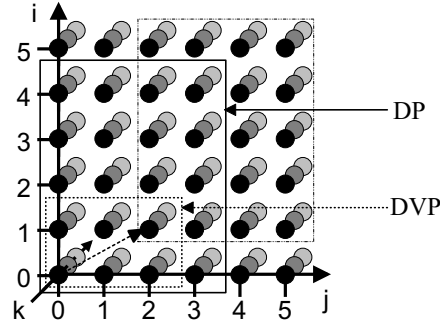


Fig. 9. DP and DVP of the dependency between $A(0)$ and $B(0)$ of example in Figure 1 with dimension k fixed innermost.

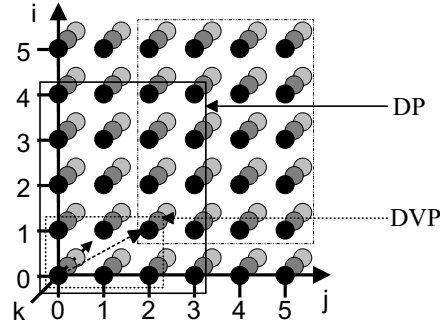


Fig. 10. DP and DVP of the dependency between $A(0)$ and $B(0)$ of example in Figure 1 with dimension k fixed outermost.

after the fixation of the k dimension outermost, SD i is fixed second outermost. According to the principle outlined in Table I, all remaining SDs in the DVP, only j in this example, are extended to the boundary of the DP. This fixation also result in removal of DP-elements with i dimension values larger than SV_i . The resulting DP and DVP are shown in Figure 11 and the estimated bounds are now:

$$UB = \text{size}(\text{DP}) - \text{size}(\text{DP}_{\text{overlap}}) = 8 - 2 = 6$$

$$LB = \text{size}(\text{DVP}) - \text{size}(\text{DVP}_{\text{overlap}}) = 8 - 2 = 6$$

For the execution orderings used in the illustrations above, the simple estimation principles outlined in Table I end up with converging and true upper and lower bounds. This is however not always the case. [Kjeldsberg et al. 2003] presents more complex, but still not computationally hard, calculation techniques required to ensure converging and true bounds. These techniques use a number of guiding principles that have been derived for the best-case and worst-case ordering of the unfixed dimensions. If the main goal is to reduce the storage requirement through optimization of the in-place mapping opportunity, SDs should for example be fixed innermost while NDs are fixed outermost. More complex mathematical reasoning lies behind similar rules for the internal ordering among SDs. The best case and worst-case execution orderings found with these guiding principles are then used to calculate the lower and upper bounds on the dependency's storage requirement re-

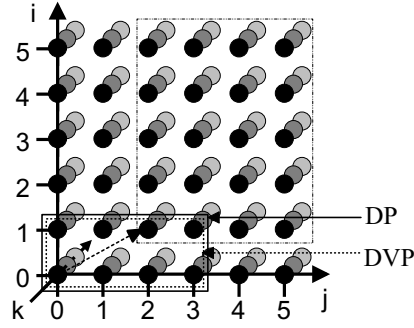


Fig. 11. DP and DVP for the dependency between $A(0)$ and $B(0)$ of example in Figure 1 with dimension k fixed outermost and i fixed second outermost.

spectively. When a few major dependencies determine the total storage requirement of an application, the guiding principles can also be used directly to determine the optimal execution ordering. For the more general case with multiple and possibly contending optimal orderings, an automated tool based on the estimation results is required. Such a tool will be presented in Section 4. The details and proofs of the guiding principles can be found in [Kjeldsberg 2001]. They are not the main focus of this paper however. More information will be presented in a future journal submission.

3.5 Estimation of global 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 detect which of the dependencies that are alive simultaneously. Their combined size gives the current storage requirement at any point during the execution of the application.

DEFINITION 3.2. *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 which dependencies that are actually simultaneously alive for a given partially fixed execution ordering. When multiple dependencies cover the same array elements, this is taken into account during the calculation of the combined size of the dependencies found to be alive simultaneously. Details regarding this part of the methodology are presented in [Kjeldsberg et al. 2001]. It will not be elaborated further in this paper.

4. PROTOTYPE TOOL FOR STORAGE REQUIREMENT ESTIMATION AND OPTIMIZATION

A prototype CAD tool, STOREQ, for STORage REquirement estimation and optimization, has been developed to prove the feasibility and usability of the methodology. The current version has been developed using MATLAB [The Math Works Inc. 1999], and must be run on the MATLAB platform. The plans for future work include a tighter integration of the STOREQ functionality into the ATOMIUM tool set on top of the ATOMIUM Polyhedral Dependency Graph kernel. A first prototype of an interface between ATOMIUM and STOREQ has already been developed at IMEC. This section gives a short description of the functionality of the tool, and presents some experimental results demonstrating the run time complexity. Section 5 gives examples of how the tool has been used during data transfer and storage exploration on real life applications.

4.1 STOREQ Functionality

The STOREQ program performs storage requirement estimation and optimization given a polyhedral description of a data intensive application, and a possibly partially fixed execution ordering. The implemented methodologies are based on the principles presented in Section 3, but also include the more advanced calculation technique presented in [Kjeldsberg et al. 2003].

The main input to STOREQ is two matrixes defined using a simple text format. One matrix defines the polytopes in the common iteration space in which the statements of the application produce array elements. That is, their iteration domains. This matrix is denoted the *Set Of Statement Polytopes* (SetOfSP). The second matrix defines the dependencies between these polytopes and is denoted the *Set Of Dependencies* (SetOfDep). Figure 12 gives examples of the format of these matrixes. In addition, two more input matrixes are automatically generated by the program, which can afterwards be changed by the user before parts of STOREQ is rerun. These matrixes are the *Fixed Dimensions* (FD) matrix and the *Ignore Dependency* (IgnoreDep) matrix. The main output from STOREQ is formed by two matrixes. One presents the estimated upper and lower bounds on the storage requirement for each dependency. This matrix is denoted the *Set Of Bounds* (SetOfBounds). The second matrix presents the optimal ordering of dimensions for each dependency as found by the program, and is denoted the *Set Of Fixed Dimensions Best Case Ordering* (SetOfFDBC). In addition, the program provides guiding hints to the user regarding how to organize the dimensions legally for negative dependencies, and a suggestion for the next FD to use.

As can be seen from the input and output description, the estimation part of the current tool version focuses on individual dependencies. Detection of simultaneously alive dependencies is not included and a total storage requirement estimate is not produced. The optimization part of the tool has a more global view. When the suggestion for the next FD to use is selected, all dependencies are taken into account in the following way:

- Detect dimensions that are NDs for all dependencies. These dimensions should be fixed outermost.
- For each dimension, count the number of dependencies for which it is an SD.

```

for (i=0; i<=7; i++)
  for (j=0; j<=7; j++) {
    A[i][j] = f1( in );
    if (i >= 1) C[i][j][k] = B[i][j] = f2( A[i-1][j] );
  }

SetOfSP = [ 0 -7 0 -7; ...
           1 -7 0 -7];
SetOfDep = [ 1 2 1 0 -1 0 -0];

```

Fig. 12. Reference application code and input matrices for STOREQ tool.

The dimension(s) with the highest numbers should be fixed innermost.

- For each nest level starting innermost, find the dimension that is an SD for most dependencies at the current or at lower nest levels, and that has not been fixed so far. This dimension should be fixed at the current nest level.

It is possible for the designer to interactively exclude dependencies from the search for the next FD to use. If, for example, the code is decided split into two loop nests, the ordering of each should be optimized without considering the dependencies of the other. This is done using the IgnoreDep matrix, where the designer specifies the dependencies to be ignored.

The execution of STOREQ is split into two parts. First, an initialization script is run that reads the input and detects negative dependencies in the code. The FD and IgnoreDep matrixes are also initialized. This is followed by the main run of STOREQ that performs the actual estimation. The main run can then be repeated with alternative combinations of dimensions fixed in the FD and/or ignored using IgnoreDep. As shown in Section 5, this approach guides the designer towards an optimized end result. More work is still needed to include the remainder of the estimation and optimization methodology into the STOREQ tool.

4.2 Run Time Experiments

A number of experiments have been performed to determine the run time complexity of the STOREQ tool. The FLOPS function and the CPUTIME function of MATLAB are used to evaluate the run time complexity. FLOPS is a floating point operation count returning the cumulative number of floating point operations. CPUTIME returns the CPU time in seconds that has been used by MATLAB since MATLAB was started. A very simple application that is partly parameterizable is used as reference for the more complex experiments. The common iteration space of this application has two dimensions (each running from 0 to 7), one Spanning Dimension (SD), and no fixed dimensions. The application code and the corresponding input matrixes are shown in Figure 12. Each row of SetOfSP defines the iteration domain (or statement polytope) of one statement. Each row of SetOfDep defines one dependency where the first two columns define between which two rows in SetOfSP the dependency runs. The third column defines the number of Dependency Vectors (DVs) in the dependency, while the remaining columns contain the actual DVs. More than one DV is needed to represent all extreme DVs if the dependency is non-uniform.

Table II lists the FLOPS and CPUTIME results for a number of different experiments performed on a 200MHz HP K-370 machine with PA-RISK 8200 processors, 4MB cache, and 3GB RAM. The columns of the table should be understood as follows:

#Dim is the number of dimensions in the common iteration space, and for these experiments also the number of dimensions in each dependency.

Max iter value is the maximal iterator value for the dimensions. It thus defines the size of the iteration space.

#Dep is the number of dependencies in the iteration space.

Length DV is the length of the Dependency Vector (DV) of the dependencies.

#SD is the number of Spanning Dimensions (SDs) in the dependencies.

#FD is the number of fixed dimensions.

#FLOPS / CPUTIME[s] Init is the number of flops and the cpu time reported by MATLAB while running the initialization part of STOREQ.

#FLOPS / CPUTIME[s] Run is the number of flops and the cpu time reported by MATLAB while running the main body of STOREQ.

A number of interesting results can be read from Table II. Experiment c) shows that the run time is unaffected by the size of the common iteration space. This is important, since the common iteration space may be very large, especially for multi-media applications. Experiment e) shows that the length of the DV does not influence the run time either, while b) shows that the runtime is less than linearly dependent on the number of dimensions. A linear dependency would be expected in b), but due to overhead not related to the number of dimensions, this is not so. As experiment f) shows, the run time is more affected by the number of SDs since they require a more complex calculation for their internal ordering so that the non-dimensional overhead is less dominant. Experiment d) shows that the run time is only linearly dependent on the number of dependencies in the common iteration space. This is also the case when the number of SDs is increased, as illustrated in Experiment f). Experiment h) and j) demonstrate how the number of fixed dimensions is important since it requires a somewhat more complex handling than unfixed dimensions. The complexity is still polynomial with respect to the combination of dependencies, SDs and fixed dimensions. This is important to avoid an explosion of the time complexity as the number of dependencies increase.

In general, the experiments show run times in accordance with theoretical time complexity for the estimation methodology discussed in [Kjeldsberg 2001]. The actual cpu time used for each experiment is not important, except for comparisons between the examples. Even though all experiments already have cpu times of less than one second, these times will be even further decreased when a stand alone and optimized final tool is built instead of using a prototype that runs on the MATLAB platform. With these runtimes the estimation techniques can easily be included in the core of other optimization tools. This is crucial because many system-level transformation steering engines require such early estimates on bounds. See e.g. [Catthoor et al. 1998] for examples of memory related optimization steps.

Table II. Run time complexity of STOREQ tool.

	#Dim	Max iter value	#Dep	Length DV	#SD	#FD	#FLOPS/ CPUTIME[s] Init	#FLOPS/ CPUTIME[s] Run
a)	2	7	1	1	1	0	27/0.00	213/0.03
b)	6	7	1	1	1	0	79/0.00	425/0.04
c)	2	1023	1	1	1	0	27/0.00	213/0.03
d)	2	7	10	1	1	0	261/0.01	2121/0.23
e)	2	7	1	3	1	0	27/0.00	213/0.03
f)	6	7	1	1	4	0	79/0.00	638/0.05
g)	2	7	1	1	1	2	27/0.00	277/0.05
h)	6	7	1	1	4	6	79/0.00	825/0.09
i)	6	7	10	1	4	0	781/0.01	6371/0.37
j)	6	7	10	1	4	6	781/0.01	8511/0.84

5. ESTIMATION ON REAL LIFE APPLICATION DEMONSTRATORS

In this Section the usefulness and feasibility of the estimation methodology and the STOREQ CAD tool is demonstrated using real life applications.

5.1 Cavity detection algorithm

5.1.1 Code description. The estimation methodology has been applied to a cavity detection algorithm used for detection of cavity perimeters in medical imaging. Figure 13 shows the important parts of the code. A pseudo t dimension is inserted to generate a common iteration space. As a starting point, the pseudo t dimension is defined so that statements placed in separate loop nests in the original code are executed sequentially. The corresponding data flow graph is given in Figure 14. Each node depicts the *iteration domain* (ID) of a statement (also called statement polytope). The edges represent dependencies between the IDs. The dependencies are enumerated to ease the following discussion. The vertical lines divide the IDs into groups that are executed at different iterator values of the t dimension. The t dimension opens for transformations such as loop merging. The storage requirement of the transformed code with alternative placements and orderings can thus be compared to the storage requirement of the original ordering. In this example, the focus is on parts with major impact on the storage requirement, so boundary conditions are ignored. It is assumed that the input image can be presented at the input of the application in any order.

The STOREQ CAD tool has been used extensively during the exploration of the cavity detection algorithm. As allowed by the tool, some of the dependencies in the code are non-uniform. Up to four extreme DVs are needed for their description. Statement S.3 has for example a non-uniform accessing of array `gauss_x_image`. Depending on the k value, an array element is accessed that was produced at an iteration point with a y value smaller, equal, or larger than the current y value. This gives rise to two extreme DVs, one in each direction along the y dimension and with different lengths in the k dimension. Furthermore, a number of negative dependencies exist in the code. Table III summarizes the negative dimensions, the SDs, and the NDs for each DV of the dependencies. This information is automatically dumped from the STOREQ tool. The dependency enumeration is in

```

#define GB 1
#define TOT (2*GB)+1
#define NB 8
#define N 480
#define M 640
int x_offset[NB]={1,1,1,0,0,-1,-1,-1};
int y_offset[NB]={1,0,-1,1,-1,1,0,-1};
for (t=0; t<=5; t++)/* Dimension 1 */
  for (x=0; x<=N-1; x++) /* Dimension 2 */
    for (y=0; y<=M-1; y++) /* Dimension 3 */
      for (k=0; k<=NB-1; k++) { /* Dimension 4 */
S.1      if (t==0 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB & k<=TOT-1)
          gauss_x_compute[x][y][k+1] = f1(gauss_x_compute[x][y][k], image_in[x+k-1][y]);

S.2      if (t==0 & k==TOT-1 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          gauss_x_image[x][y] = f2(gauss_x_compute[x][y][TOT]);

S.3      if (t==1 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB & k<=TOT-1)
          gauss_xy_compute[x][y][k+1] = f3(gauss_xy_compute[x][y][k],
                                             gauss_x_image[x][y+k-1]);

S.4      if (t==1 & k==TOT-1 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          gauss_xy_image[x][y] = f4(gauss_xy_compute[x][y][TOT]);

S.5      if (t==2 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          maxdiff_compute[x][y][k+1] = f5(gauss_xy_image[x+x_offset[k]][y+y_offset[k]],
                                             gauss_xy_image[x][y], maxdiff_compute[x][y][k]);

S.6      if (t==2 & k==NB-1 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          comp_edge_image[x][y] = f6(maxdiff_compute[x][y][NB]);

S.7      if (t==3 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          out_compute[x][y][k+1] = f7(comp_edge_image[x+x_offset[k]][y+y_offset[k]],
                                       comp_edge_image[x][y], out_compute[x][y][k]);

S.8      if (t==3 & k==NB-1 & x>=GB & x<=N-1-GB & y>=GB & y<=M-1-GB)
          image_out[x][y] = f8(out_compute[x][y][NB]);
      }
}

```

Fig. 13. Code for Cavity Detection example.

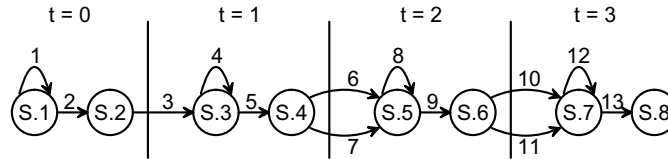


Fig. 14. Data-flow graph for Cavity Detection example.

Table III. NDs (indicated with -), SDs, and negative dimensions (underlined) for each DV of the dependencies in the cavity detection code.

Dep.	DV1	DV2	DV3	DV4
1	-t -x -y k			
2	-t -x -y -k			
3	t -x y <u>k</u>	t -x <u>y</u> -k		
4	-t -x -y k			
5	-t -x -y -k			
6	t -x -y <u>k</u>	t -x -y k		
7	t <u>x</u> <u>y</u> <u>k</u>	t <u>x</u> y -k	t x <u>y</u> k	t x y k
8	-t -x -y k			
9	-t -x -y -k			
10	t -x -y <u>k</u>			
11	t <u>x</u> <u>y</u> <u>k</u>	t <u>x</u> y <u>k</u>	t x <u>y</u> <u>k</u>	t x y -k
12	-t -x -y k			
13	-t -x -y -k			

accordance with Figure 14.

5.1.2 *Storage requirement estimation and optimization.* It will now be shown how the STOREQ tool can be used to rearrange the common loop nest and to determine an execution ordering that gives an optimized storage requirement for the application. The first run of STOREQ without any ordering fixed shows that all dependencies that do not cross a vertical t -line in Figure 14, has the k dimension placed innermost in its optimal ordering. This is due to the fact that all other dimensions are NDs for these dependencies. They should hence be fixed outside the k dimension according to the guiding principles outlined in the last paragraph of Section 3.4.

For dependencies with t as an SD, that is all dependencies crossing a t -line in Figure 14, the upper and lower bounds converge at the previous upper bound if t is fixed outermost. This is again in accordance with the guiding principles which indicate a large penalty on placing SDs outermost. Due to the loop splitting caused by the t dimension, none of these dependencies is alive simultaneously if t is fixed outermost. Their individual sizes are hence determining the overall storage requirement. Since the t dimension is an artificial dimension used for generation of a common loop nest, it must always be fixed outermost. To reduce the storage requirement, the common iteration space must therefore be rearranged so that the dependencies do not have t as an SD (do not cross a t -line). The removal of the t dimension from the set of SDs for dependencies corresponds to a loop merging. This may cause dependencies to be alive simultaneously, so that their combined sizes determine the global storage requirement.

The first objective is to reposition statement S.3 in the common iteration space so that the t dimension becomes an ND for dependency 3. This corresponds to a loop merging between the first and second loop nest in the original code. Dependency 3 has two extreme DVs, as seen in Table III. Employing techniques from the Data Transfer and Storage Exploration methodology formalized in [Catthoor et al. 1998], it can be found that a legal loop merging can be performed if the y dimension is fixed outside the k dimension in the merged loop nest. In addition, the ID of S.3 must be moved one iteration node up along the y dimension to make it non-negative

```

if(t==0 & x>=GB & x<=N-1-GB & y>=GB+1 & y<=M-1-GB+1 &
                                     k<=TOT-1)
S.3    gauss_xy_compute[x][y-1][k+1] = f3(gauss_xy_compute[x][y-1][k],
                                     gauss_x_image[x][y-1+k-1]);

```

Fig. 15. Statement S.3 after transformation.

Table IV. Estimated dependency sizes resulting from alternative transformations ($N = 480$, $M = 640$). UB = LB where only one number is reported.

		Dep. 3	Dep. 7	Dep. 11	Combined
a)	Original code (FD = [1 X X X])	304964	304964	304964	304964
b)	y positive (FD = [1 3 2 4])	956	1435	1435	3826
c)	x positive (FD = [1 2 3 4])	2	1915	1915	3832
	No neg. dim. (FD = [1 X X 4])	2/956	959/1279	959/1279	1920/3514
d)	No neg. dim. (FD = [1 3 2 4])	956	959	959	2874
e)	No neg. dim. (FD = [1 2 3 4])	2	1279	1279	2560

in DV2 of dependency 3. A form of *skewing* [Kulkarni and Stumm 1993] is used, and the corresponding statement after the move is given in Figure 15. Running STOREQ reveals that the lower bound of dependency 3 is now increased from 1 to 2 while the upper bound is reduced from 304964 to 956 since the DP of S.2 and its depending iteration nodes are now partially overlapping.

The remaining dependencies with t as SD are dependencies 6, 7, 10, and 11. A number of transformations exist that fulfills the requirements of a legal full loop merging. Each of these has different effects on the size of these and other dependencies. The STOREQ tool has been used to evaluate the global consequences of a number of these alternative transformations. In addition to fixation of dimensions in a given order in the merged loop nest, the IDs have been repositioned through skewing in a similar way as discussed above. Table IV presents the STOREQ estimation results for dependencies 3, 7, and 11. The elements carried by dependency 6 (10) is a subset of those carried by dependency 7 (11), so only dependency 7 (11) needs to be included in the combined size. Each row shows the estimated storage requirement for alternative legal placements and execution orderings in this new common iteration space. The storage requirement of the original code without loop merging is shown for comparison in the first row. Row e) holds the globally best solution. It has a storage requirement over two orders of magnitude lower than the original solution, and substantially lower than the other alternatives.

If the same experiment is performed using a different image format, for instance the panoramic format of the *Advanced Photo System*, the conclusion turns out to be somewhat different. The previously best solution is now second to worst as shown in Table V and Figure 16. The storage required for buffering full image lines along the y dimension for dependency 7 and 11 are now larger than that of buffering full image lines along the x dimension for dependency 3. These somewhat surprising results demonstrate how important the storage requirement estimation tool is for the optimization of the memory usage.

Table V. Estimated dependency sizes resulting from alternative transformations for another image size ($N = 480$, $M = 1380$). UB = LB where only one number is reported.

		Dep. 3	Dep. 7	Dep. 11	Combined
a)	Original code (FD = [1 X X X])	658684	658684	658684	658684
b)	y positive (FD = [1 3 2 4])	956	1435	1435	3826
c)	x positive (FD = [1 2 3 4])	2	4135	4135	8272
	No neg. dim. (FD = [1 X X 4])	2/956	959/2759	959/2759	1920/6474
d)	No neg. dim. (FD = [1 3 2 4])	956	959	959	2874
e)	No neg. dim. (FD = [1 2 3 4])	2	2759	2759	5520

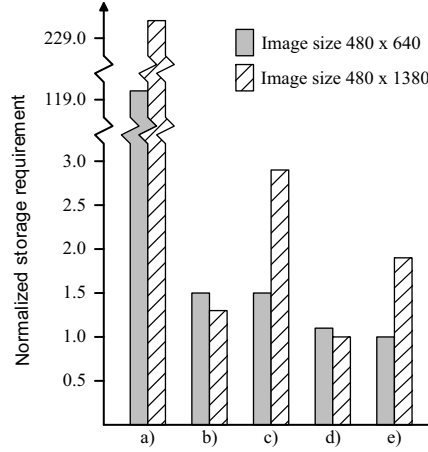


Fig. 16. Combined storage requirement for alternative cavity detection implementations. Normalized to best solution for each image size. Transformations as in Table V.

5.2 MPEG-4 motion estimation kernel

MPEG-4 is a standard from the *Moving Picture Experts Group* for the format of multi-media data-streams in which audio and video objects can be used and presented in a highly flexible manner [The ISO/IEC Moving Picture Experts Group 2003; Sikora 1997]. An important part of the coding of this data-stream is the motion estimation of moving objects. See [Brockmeyer et al. 1999] for a more detailed description of the motion estimation part of the standard. Figure 17 summarizes the estimation results found using the STOREQ tool during the early steps of the design trajectory. More details can be found in [Kjeldsberg et al. 2001].

The leftmost column shows the combined declared size of the two most important arrays in this part of the code. This is the memory size (262400) the designer would have to assume if no estimation and optimization tools were available. The next column shows the result found using the estimation technique described in [Balasa et al. 1995] (45296). This result will be the same independently of the execution ordering. Finally the columns marked a) through e) show the size estimates found for a number of partially fixed execution orderings using the methodology presented in this paper. Both results for the combined size of individual dependencies and the size of simultaneously alive dependencies are reported. With no execution ordering fixed, the UB on the combined size of individual dependencies is 51457 while the simultaneously alive dependencies have an UB on the combined size of 15616. This

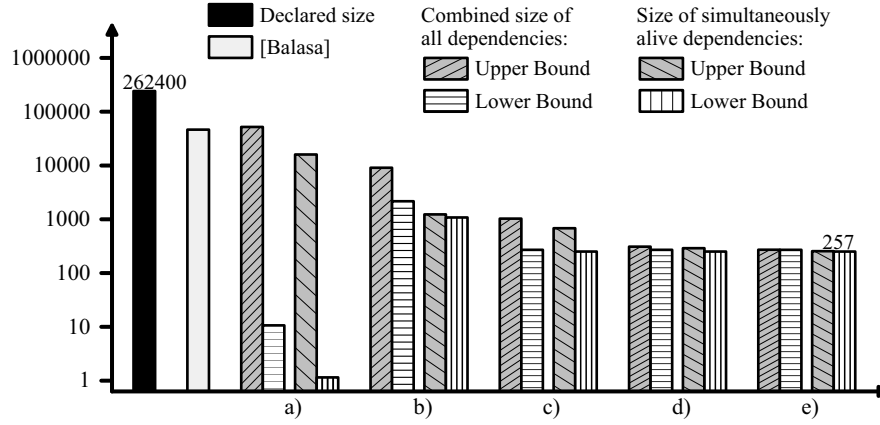


Fig. 17. Estimation results for MPEG-4 motion estimation using various techniques and partially fixed execution orderings. a) No ordering, b) $y-p$ outermost, c) $y-s$ outermost, d) $x-s$ second outermost, e) $y-p$ third outermost (fully fixed).

shows the importance of both steps in the estimation methodology. In Figure 17 b) one dimension is fixed outermost while the remaining dimensions are still unfixed. This results in a decrease in the upper bound (combined size 1280) and an increase in the lower bound (combined size 1025) compared to the fully unfixed ordering in a). In c) an alternative dimension is fixed outermost. This results in a much larger decrease in the upper bound (combined size 736) while the increase in the lower bound is much smaller (combined size 257). Even with such limited information available it is possible for the designer to conclude that the outer dimension used in c) is better than the one used in b). Using the guiding principles and feedback from the estimation tool, the designer is finally able to reach a storage requirement of 257 when the full execution ordering is fixed.

6. CONCLUSION

We have presented a storage requirement estimation methodology to be used during the early system design steps. The execution ordering is at this point typically partially fixed, and this is taken into account to produce upper and lower bounds on the storage requirement of the final implementation. The methodology is divided into four steps. In the first step, a data-flow graph is generated that reflects the data dependencies in the application code. The second step places the polyhedral descriptions of the array accesses and their dependencies in a so-called common iteration space. The third step estimates the upper and lower bounds on the storage requirement of individual data dependencies in the code, taking into account the available execution ordering. As the execution ordering is gradually fixed, the upper and lower bounds on the data dependencies converge. This is a very useful and unique property of the methodology. Finally, simultaneously alive data dependencies are detected. Their maximal combined size at any time during execution equals the total storage requirement of the application. A new prototype CAD tool has been presented that includes major parts of the storage requirement estimation and optimization methodology. Using manually generated and real life design ex-

amples the tool proves the feasibility of the techniques and in particular shows that run times on computers will be short, in the order of seconds even for substantial applications.

ACKNOWLEDGMENTS

The work in this paper was supported in part by the Norwegian Research Council through research project 131359 CoDeVer.

REFERENCES

- BALASA, F., CATTHOOR, F., AND DE MAN, H. 1995. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on VLSI Systems* 3, 2 (June), 157–172.
 - BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Acad. Publ., Boston, USA.
 - BORMANS, J., DENOLF, K., WUYTACK, S., NACHTERGAELE, L., AND BOLSENS, I. 1999. Integrating system-level low power methodologies into a real-life design flow. In Proc. Wsh. on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE, Kos, Greece, 19–28.
 - BROCKMEYER, E., NACHTERGAELE, L., CATTHOOR, F., BORMANS, J., AND DE MAN, H. 1999. Low power memory storage and transfer organization for the mpeg-4 full pel motion estimation on a multi media processor. *IEEE Trans. on Multimedia* 1, 2 (June), 202–216.
 - CATTHOOR, F., WUYTACK, S., DE GREEF, E., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Acad. Publ., Boston, USA. ISBN 0-7923-8288-9.
 - CHAKRABARTI, C. 2001. Cache design and exploration for low power embedded systems. In Proc. Intl. Conf. on Performance, Computing, and Communications. IEEE, Phoenix, Arizona, USA, 135–139.
 - DANCKAERT, K. 2001. Loop transformations for data transfer and storage reduction on multiprocessor systems. Ph.D. thesis, ESAT/EE Dept., K.U.Leuven, Leuven, Belgium.
 - DANCKAERT, K., CATTHOOR, F., AND DE MAN, H. 2000a. A loop transformation approach for combined parallelization and data transfer and storage optimization. In Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications, PDPTA'00. Las Vegas NV, USA, 2591–2597.
 - DANCKAERT, K., CATTHOOR, F., AND DE MAN, H. 2000b. A preprocessing step for global loop transformations for data transfer and storage optimization. In Proc. Intl. Conf. on Compilers, Arch. and Synth. for Emb. Sys. San Jose, CA, USA, 34–40.
 - DE GREEF, E., CATTHOOR, F., AND DE MAN, H. 1997. Array placement for storage size reduction in embedded multimedia systems. In Proc. Intl. Conf. on Applic.-Spec. Systems Arch. and Processors. Zurich, Switzerland, 66–75.
 - FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Intl. J. of Parallel Programming* 20, 1, 23–52.
 - GAJSKI, D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and design of embedded systems*. Prentice Hall, Englewood Cliffs NJ, USA.
 - GEBOTYS, C. H. AND ELMASRY, M. I. 1991. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In Proc. of the 28th ACM/IEEE Design Automation Conf. San Jose CA, USA, 2–7.
 - GRUN, P., BALASA, F., AND DUTT, N. 1998. Memory size estimation for multimedia applications. In Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes). Seattle WA, USA, 145–149.
 - IMEC. 2003. Atomium web site. <http://www.imec.be/design/multimedia/atomium/>.
 - KIROVSKI, D., LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1999. Application-driven synthesis of memory-intensive systems-on-chip. *IEEE Trans. on Comp.-aided Design* 18, 9 (Sept.), 1316–1326.
- ACM Transactions on Design Automation of Electronic Systems, Vol. V, No. N, MM 20YY.

- KJELDSBERG, P. G. 2001. Storage requirement estimation and optimisation for data-intensive applications. Ph.D. thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway. ISBN 82-7984-174-1.
- KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2000a. Automated data dependency size estimation with a partially fixed execution ordering. In Proc. IEEE Intl. Conf. on Comp. Aided Design. Santa Jose CA, USA, 44–50.
- KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2000b. Storage requirement estimation for data-intensive applications with partially fixed execution ordering. In Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (Codes). San Diego CA, USA, 56–60.
- KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2001. Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications. In 38th ACM/IEEE Design Automation Conf. Las Vegas N, USA, 365–370.
- KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2003. Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications. Accepted for publication in *IEEE Trans. on Comp.-aided Design*.
- KULKARNI, D. AND STUMM, M. 1993. Loop and data transformations: A tutorial. Tech. Rep. CSRI-337, Computer Systems Research Inst., Univ. of Toronto. June.
- KURDAHI, F. J. AND PARKER, A. C. 1987. Real: a program for register allocation. In Proc. 24th ACM/IEEE Design Automation Conf. Miami FL, USA, 210–215.
- LEFEBVRE, V. AND FEAUTRIER, P. 1997. Optimizing storage size for static control programs in automatic parallelizers. In Proc. EuroPar Conf. Lecture notes in computer science, vol. 1300. Springer Verlag, Passau, Germany, 356–363.
- OHM, S. Y., KURDAHI, F. J., AND DUTT, N. 1994. Comprehensive lower bound estimation from behavioral description. In IEEE/ACM Intl. Conf. on Computer-Aided Design. IEEE, San Jose CA, USA, 182–187.
- PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1999. Local memory exploration and optimization in embedded systems. *IEEE Trans. on Comp.-aided Design* 18, 1 (Jan.), 3–13.
- PAULIN, P. G. AND KNIGHT, J. P. 1989. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Trans. on Comp.-aided Design* 8, 6 (June), 661–679.
- PUGH, W. AND WONNACOTT, D. 1993. An exact method for analysis of value-based array data dependences. In Proc. 6th Intl. Wsh. on Languages and Compilers for Parallel Computing. Portland OR, USA, 546–566.
- QUILLERE, F. AND RAJOPADHYE, S. 2000. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems* 22, 5 (Sept.), 773–815.
- RAMANUJAM, J., HONG, J., KANDEMIR, M., AND NARAYAN, A. 2001. Reducing memory requirements of nested loops for embedded systems. In 38th ACM/IEEE Design Automation Conf. Las Vegas NV, USA, 359–364.
- SHANG, W., HODZIC, E., AND CHEN, Z. 1996. On uniformization of affine dependence algorithms. *IEEE Trans. on Computers* 45, 7 (July), 827–840.
- SIKORA, T. 1997. The mpeg-4 video standard verification model. *IEEE Trans. on Circuits and Systems for Video Technology* 7, 1 (Feb.), 19–31.
- THE ISO/IEC MOVING PICTURE EXPERTS GROUP. 2003. Mpeg web page. <http://mpeg.telecomitalialab.com/>.
- THE MATH WORKS INC. 1999. *MATLAB, The Language of Technical Computing, Using MATLAB Version 5*. Natick, MA, USA.
- TSENG, C.-J. AND SIEWIOREK, D. 1986. Automated synthesis of data paths in digital systems. *IEEE Trans. on Comp.-aided Design* 5, 3 (July), 379–395.
- VERBAUWHED, I., SCHEERS, C., AND RABAEY, J. 1994. Memory estimation for high-level synthesis. In Proc. 31st ACM/IEEE Design Automation Conf. San Diego CA, USA, 143–148.
- WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. The Addison-Wesley Publishing Company, Redwood City CA, USA. ISBN 0-8053-2730-4.
- ZHAO, Y. AND MALIK, S. 1999. Exact memory size estimation for array computation without loop unrolling. In Proc. 36th ACM/IEEE Design Automation Conf. New Orleans LA, USA, 811–816.