Inter in-place storage size requirement estimation

P. Rydland[1], M. Palkovic[2], P.G. Kjeldsberg[1], E. Brockmeyer[2], F. Catthoor[2,3]
[1]Norwegian University of Science and Technology, Trondheim, Norway (email: pgk@fysel.ntnu.no)
[2]IMEC, Leuven, Belgium [3]also at Katholieke Universiteit Leuven, Belgium (email: catthoor@imec.be)

Abstract

Data storage is an important contributor to power dissipation, particularly in multi-media embedded systems. To achieve low power implementation for these systems, the storage requirement can be reduced by exploiting limited data life-times and reuse of memory locations. This optimisation can only be performed quite late in the design trajectory because most of the decisions on e.g. detailed memory access organisation should have been fixed. Thus an early and fast estimate of the storage requirement is crucial for the system designer. This paper proposes a grouping technique that identifies simultaneously alive sets of data dependencies. The grouping technique extends previous work on storage size requirement estimation for individual data dependencies, allowing the designer to obtain the global storage size requirement for an application. A prototype CAD tool has been developed and used to test the technique on real-life multi-media kernels.

1 Introduction

Power consumption is an important design parameter for portable electronic devices. Since many such systems are powered by batteries, high power consumption leads to lower availability. High power dissipation also increases the need for cooling and packaging. For data dominated embedded applications such as embedded multi-media and signal processing, memory accesses are the major contributors to power dissipation. Since large memories are needed, and their accessing is a dominant speed bottleneck, chip size and system performance are also highly influenced by the overall memory management [1].

For the target classes of data dominant applications the high level description is typically characterised by large multi-dimensional loop nests and arrays. Alternative implementations of a given application may have very different data storage requirements for these arrays [2]. A deciding factor for the required storage size is the possibility for reusing memory locations. Memory locations can be reused for data with non-overlapping life-times as shown in Figure 1. 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. Mapping the different data to the same physical memory location is called in-place mapping [3]. We distinguish between two types of in-place mapping, namely intra-array in-place and inter-array in-place. During intra-array in-place mapping the different elements of the same array are mapped to the same memory location. During inter-array in-place mapping the different arrays are mapped to the same memory location. The intra- and inter-array in-place mapping is only performed as one of the last steps of the system design trajectory. Many early design steps, e.g. global data flow and loop transformations, may however greatly effect the in-place mapping opportunities. It is consequently essential to give the designer estimates of the resulting memory size requirements very early in the design trajectory.

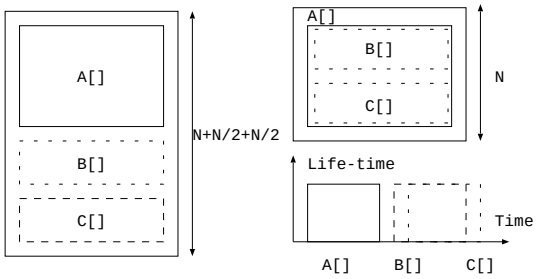

Figure 1: Illustration of inter in-place optimisation

This paper presents a grouping technique for memory size estimation and is an extension of previous work by Kjeldsberg et. al. [4]. Section 2 gives an introduction to the polytope model that is being used both in the previous work and here. In Section 3 a grouping technique that allows an efficient inter-array in-place mapping, is detailed. In this Section also the original technique in the context of our extension is recapped. This is followed by a description of results from applying our technique on a real-life multi-media application kernel in Section 4. In Section 5 the new technique is compared to previous work, before conclusions are given in Section 6.

2 Polytope Model

The Polytope model used in this paper is an extension of the model described in [4]. The model is outlined in the rest of this Section together with the necessary extensions used later.

At a high abstraction level, data intensive applications can be modelled as large multi-dimensional arrays and deeply nested loops. It is possible to transform this program code to a perfectly nested loop, which means that all statements are performed in the innermost loop [5]. The perfectly nested loop forms a mathematical space called iteration space $\mathbb{Z}^n$, where the iterations are modelled as iteration nodes. Sets of iteration nodes where the statements are executed can be represented by (unions of) polytopes called iteration domains(ID). As an example, the Polytope model extracted from the C-code found in Figure 2 is shown in Figure 3.

When a data element is produced and later consumed a data dependency exists between the producing and consuming statement. The lifetime of the data dependency reflects the period of time during which the data

```
for (int i=0;i<=9;i++) {
  for (int j=0;j<=9;j++) {
    if ( 1 <= i <= 5 ) && ( 2 <= j <= 4 )
      A[i][j] = a(); // statement A
    if ( 7 <= i <= 9 ) && ( 2 <= j <= 4 )
      B[2*i][j] = b(); // statement B
    if ( 1 <= i <= 5 ) && ( 6 <= j <= 8 )
      C[i+j][j] = c();  // statement C
  }
}
```

Figure 2: Code example

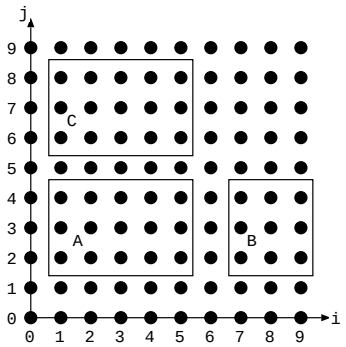

Figure 3: An iteration space and three statement polytopes

element needs to be stored in memory. From this the strong relationship between dependencies and memory storage requirements is clear. Therefore, it is vital to be able to model data dependencies.

A node dependency between two nodes means that data produced at the first node is consumed at the second node. The dependency part (DP), see Figure 5, contains the iteration nodes where array elements are produced that are read within the depending iteration domain (DID). The dependency between a DP and its corresponding DID is a domain dependency. A domain dependency can be represented as a domain dependency vector starting at the node with the least lexicographical value in the DP and ending at the node with the least lexicographical value in the DID. If the node dependency vectors are uniform, the domain dependency vector will be equal to any of the node dependency vectors. In case the node dependencies are non-uniform, the node dependencies must be transformed into uniform dependencies. This can be done using the concept of extreme dependencies [6].

The size of a domain dependency is a function of its position in the iteration space. For nodes where the dependency is not alive the dependency size is zero, for other nodes the domain dependency size is the number of alive node dependencies at that particular iteration node related to that particular domain dependency. Dimensions where the domain dependency has a length different from zero, are called spanning dimensions (SD). All other dimensions are denoted non-spanning dimensions (ND). An extended dependency part (EDP) is obtained by extending the DP with the length of the domain dependency vector for all SDs. Figure 4 shows a code example where the production and consumption of A-array elements give rise to a dependency between the two statements. Figure 5 gives the corresponding iteration space with EDP, DP, DID and domain dependency vector indicated.

```
for (int i=0;i<=9;i++) {
  for (int j=0;j<=9;j++) {
    if ( 1 <= i <= 4 ) && ( 2 <= j <= 6 )
      A[i][j] = a(); // DP
    if ( 3 <= i <= 6 ) && ( 4 <= j <= 8 )
      B[i][j] = b(A[i-2][j-2]); // DID
  }
}
```

Figure 4: Code example for domain dependency

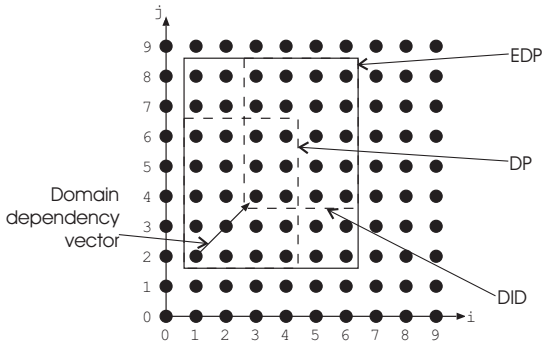

Figure 5: An EDP consists of a DP and a DID

3 Memory Size Estimation

At the core of the memory size estimation problem is the fact that data elements can be mapped in-place of each other in memory. This means that counting the size of all arrays present in the input code could lead to a huge overestimate. Depending on whether we consider first intra and then inter in-place or both intra and inter in-place at the same time we can distinguish two different approaches for in-place. The simple approach is first to perform intra in-place and reserve a constant set of locations for each array. These memory locations will be set aside for an array as long as any data element belonging to the array is alive. The size of the set must be big enough to be able to store the maximum number of simultaneously alive data elements belonging to the array. To find the storage requirement of an application at any given time during execution, the reserved sizes of the currently alive arrays are summed. Using our technique described later in this Section we can easily estimate this simple approach of in-place.

The complex approach considers both inter and intra in-place at the same time. This means we take into account that the size of the set of simultaneously alive data elements within an array varies during the execution of the application. Thus we always let the size of the set of memory locations reserved for one particular array be equal to the current set of simultaneously alive data elements. This approach is more accurate, but is complex to implement [3]. It will be studied in future work.

The goal of the memory size estimation is to identify the maximum number of alive array elements at any given time. This paper will present a methodology for memory estimation that considers the effect of intra-array in-place followed by inter-array in-place. Such an estimation is realistic for the designer as we have seen

from in-place tools used at the end of design trajectory [3]. This simple approach is mainly used because of its simple implementation.

The maximum size of a individual domain dependency can be found using the formula presented in [4]. By grouping simultaneously alive domain dependencies, and then adding their maximum domain dependency sizes together, it is possible to make an estimate that takes both intra-array and inter-array in-place into account [7]. The pseudo code listed in Figure 6 shows our technique at the top level. First `max_size` is initialised, which will at the end of the execution contain the global memory size requirement estimate. Next simultaneously alive domain dependencies are grouped. This technique will be elaborated in Section 3.1 and is the main contribution of this paper. The domain dependency size technique (line 6 in Figure 6) is reused from the previous work and shortly explained in Section 3.2.

Unlike the approach presented in [4], the iteration order is assumed to be fixed. Iteration nodes are then visited in the order given by a procedural execution of the corresponding application code. For the iteration space in Figure 3, this corresponds to first going up along the j-axis before going out along the i-axis. The i-dimensions is then denoted the outermost dimension.

```
max_size = 0
groups = Group domain dependencies \
         that are simultaneously alive.
for each group in groups:
  size = 0
  for each domain dependency in group:
    size += Identify maximum domain \
            dependency size
  max_size = max(max_size,size)
```

Figure 6: Estimation methodology

3.1 Grouping

The grouping is a recursive technique, starting at the outermost dimension. If two domain dependencies are to be simultaneously alive, their EDPs must overlap at the outermost dimension, that is, they must share at least one coordinate value for this dimension. As a first step, EDPs that overlap in the outermost dimension are grouped, resulting in a set of groups containing possible simultaneously alive domain dependencies. The next step is to partition each group into two disjunct sets, where partition one contains domain dependencies for which the outermost dimension is an SD. The domain dependencies belonging to this first partition will for sure be simultaneously alive.

Each group found by grouping simultaneously alive dependencies belonging to partition number two, will be simultaneously alive with all dependencies in partition number one. Therefore, by adding the dependencies belonging to partition one to each subset of simultaneously alive dependencies found in partition two, the complete set of simultaneously alive domain dependencies is found. The simultaneously alive dependencies in partition two is found by repeating the whole grouping process described above, but this time for the second outermost dimension and with the dependencies in partition number two as input.

It is possible to do a trade-off between accuracy and computation speed. If the recursion is stopped after $r \leq \#dimensions$, then accuracy will be reduced and the computation speed is increased. Notice that the runtime of the algorithm depends on r and the number of overlapping EDPs and not on the number of iteration nodes in the space or on the iterator bounds.

group	MC	PySTOREQ
AB	16	15
ACD	35	32
ADE	31	29
FG	12	10
Max	35	32

Table 1: Estimation results

3.2 Dependency Size Calculation

The dependency size estimation technique presented in [4] is used to find the sizes of individual domain dependencies. This technique uses the differences between SDs and NDs to find the contribution of each dimension to the overall dependency size. Starting outermost, every ND can be ignored, since every node dependency created at a given coordinate value of such an ND will also be destroyed at the same coordinate value. When the first SD is encountered, its contribution is found by multiplying the length of the domain dependency vector in this dimension with the length of the DP in all dimensions inside the current dimension. The same way of ignoring NDs and adding size contributions for SDs is used until the innermost dimension is reached. The resulting sum equals the number of iteration nodes that are visited within the DP before the first iteration node in the DID is visited. This equals the maximum number of simultaneously alive node dependencies, end hence the size of the domain dependency.

4 Results

The grouping algorithm presented in this paper have been implemented as a Python script based prototype CAD tool, called PySTOREQ. To make the tool more compact, also necessary parts from the original individual storage requirement methodology [4, 5] have been reimplemented to PySTOREQ. PySTOREQ was measured against a Memory Compaction Tool (MC), the C++ based tool used at the end of design trajectory for in-place mapping [8].

The test case, written in C, is an implementation of QSDPCM [9], which is a video compression algorithm. For in depth information about the test case, use [10] as a reference. No regression regarding accuracy was found during testing, which means that all maximum domain dependencies was found to exactly match the expected dependency sizes.

PySTOREQ was also tested against a small artificial example found in [5, Chapter 5]. The test case was also analysed by MC. As can be seen from Table 1, PySTOREQ has the same level of accuracy as MC. Because MC uses a more conservative approach, there are some small differences between the numbers found by MC and PySTOREQ.

5 Previous Work

One of the first papers dealing with system level memory size estimation was [11]. Predecessors of this paper

had mainly dealt with scalars, which is inefficient when the source code has multi-dimensional arrays and deep loop-nests. They proposed an estimation technique where a data-flow graph is extracted from a polytope model. The maximum set of simultaneously alive dependencies was found through a traversal of the graph. Although this is theoretically possible, it can be proved that this problem belongs the to the set of NP-complete problems.

In [12], a production time axis is used to find the maximum difference between the production and consumption time for any two depending instances, giving the storage requirement for one array. The total storage requirement is the sum of the requirements for each array. Only in-place mapping internally to an array is considered, not the possibility of mapping arrays in-place of each other.

Another approach is taken in [13]. 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. From this, a memory trace of upper and lower bounding rectangles as a function of time is found with the peak bounding rectangle yielding the total storage requirement. If the difference between the upper and lower bounds for this critical rectangle is too large, the corresponding loop is split into two loops, and the estimation is rerun. In the worst-case situation a full loop unrolling is necessary to achieve a satisfactory estimate, which is unaffordable.

The work presented in [14] is based on a polytope model with fixed iteration order. The authors observe that it is too costly to measure the number of live variables after the execution of each statement. They propose a method that measure the number of simultaneously alive variables after each iteration of a loop nest. Since the number of live array elements is counted for each iteration of the innermost loop, the methodology will in fact iterate over all iteration nodes in the iteration space.

In [15], 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 exists, the maximum reference window size equals the sum of the windows for individual arrays. Inter-array in-place is consequently not considered.

The technique presented in this paper do not depend on the number of iteration nodes found in the iteration space. Instead it depends on the number of dimensions and the number of EDPs. It is therefore reasonable to assume that the methods presented in this paper will result in faster estimation compared to state of the art. This is true especially in the case where it is acceptable to trade accuracy against computation speed. In addition, the new methodology performs estimates taking into account inter-array in-place optimisation, and is able to do this at the very earliest stages of the system design trajectory.

6 Conclusions

This paper has presented a grouping technique able to identify and group simultaneously alive data dependencies in data intensive embedded applications. Combined with the previous work for measuring the required storage size of individual dependencies it is able to estimate the global storage size estimation of a given application.

References

[1] 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*. Kluwer Academic Publishers, 2002.

[2] E. Brockmeyer, L. Nachtergaele, F. Catthoor, J. Bormans, and H. De Man, "Low power memory storage and transfer organization for the mpeg-4 full pel motion estimation on a multimedia processor," *IEEE Trans. Multi-media*, vol. 1, pp. 202–216, June 1999.

[3] E. D. Greef, *Storage Size Reduction for Multimedia Applications*. PhD thesis, Katholieke Univeriteit Leuven, Januar 1998.

[4] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Data dependency size estimation for use in memory optimization," *IEEE Trans. CAD*, vol. 22, pp. 908–921, July 2003.

[5] P. G. Kjeldsberg, *Storage Requirment Estimation and Optimization for Data Intensive Application*. PhD thesis, NTNU, 2001.

[6] K. Danckaert, *Loop transformations for data transfer and storage reduction on multiprocessor systems*. PhD thesis, Katholieke Universitet Leuven, 2001.

[7] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Detection of partially simultaneously alive signals in storage requirement estimation for data intensive applications," in *Design Automation ConferenceProceedings*, pp. 365 – 370, June 2001.

[8] E. D. Greef, F. Catthoor, and H. De Man, "Array placement for storage size reduction in embedded multimedia systems," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 66 – 75, July 1997.

[9] P. Strobach, "Qsdpcm – a new technique in scene adaptive coding," in Proc. 4th Eur. Signal Processing Conf., pp. 1141–1144, September 1988.

[10] K. Danckaert, F. Catthoor, H. De Man, and K. Masselos, "Strategy for power efficient combined task and data parallelism exploration illustrated on a qsdpcm video codec," *Journal of Systems Architecture*, vol. 45, pp. 791–808, April 1999.

[11] F. Balasa, F. Catthoor, and H. De Man, "Background memory area estimation for multidimensional signal processing systems," *IEEE Trans. VLSI Systems*, vol. 3, pp. 157–172, June 1995.

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

[13] P. Grun, F. Balasa, and N. Dutt, "Memory size estimation for multimedia applications," in Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design, pp. 145–149, March 1998.

[14] Y. Zhao and S. Malik, "Exact memory size estimation for array computations," *IEEE Trans. VLSI Systems*, pp. 517–521, October 2000.

[15] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan, "Memory size estimation for multimedia applications," in 38th ACM/IEEE Design Automation Conf., pp. 359–364, June 2001.