Fast Memory Footprint Estimation based on Maximal Dependency Vector Calculation

Q. Hu* A. Vandecappelle† P. G. Kjeldsberg* F. Catthoor†‡ M. Palkovic†

* Norwegian University of Science and Technology, Trondheim, Norway {qubo.hu,pgk}@iet.ntnu.no

† IMEC vzw, Leuven, Belgium {vdcappel,catthoor,palkovic}@imec.be

‡ also professor at Katholieke Universiteit Leuven, Belgium

Abstract

In data dominated applications, loop transformations have a huge impact on the lifetime of array data and therefore on memory footprint. Since a locally optimal loop transformation may have a detrimental effect somewhere else, many alternative loop transformations need to be explored. Therefore, estimation of the memory footprint is essential, and this estimation has to be fast. This paper presents a fast array based memory footprint estimation technique based on counting of iteration nodes in an iteration domain constrained by a maximal lifetime. The maximal lifetime is defined by the Maximal Dependency Vector (MDV) of the array for a given execution ordering. We further present for the first time two approaches for calculation of the MDV: a general approach based on an ILP formulation and a novel vertexes approach when iteration domains are approximated by bounding boxes. Experiments on practical test vehicles demonstrate that the estimation based on our vertexes approach is extremely fast, on average two orders of magnitude faster than the compared approaches, while still keeping the accuracy high. This enables system-level data memory footprint exploration of many different alternative transformed program codes, within interactive time limits, and on realistic complex applications.

1 Introduction

Modern real-time multimedia and telecommunication signal processing applications typically manipulate large multi-dimensional arrays within deep loop nests. A direct translation of the corresponding application code in an embedded system results in a design with large memory banks. These result in a high production cost, they dominate the energy consumption and also create system performance bottlenecks. System level optimization techniques such as loop transformations have been presented with the goal to design an optimized memory subsystem and/or to restructure the application code to take advantage of a predefined memory subsystem [2]. These techniques explore a huge search space and in each point the effect on memory footprint has to be evaluated. Therefore, the time required for memory footprint estimation becomes critical, in addition to estimation accuracy.

Memory footprint estimation has been tackled in the past at register transfer level (RTL) for scalars [9, 5, 11]. However, these techniques break down when large multi-dimensional arrays within deep loop nests are manipulated. To overcome this, several research teams have instead developed techniques for array-based memory footprint estimation.

Most array-based memory footprint estimation techniques require that the execution ordering (the order in which loops and statements are executed) is fixed. Verbauwhede et al. propose to build a production axis for each array to model the relative production and consumption time of the individual arrays [14]. The difference between these two dates equals the number of array elements produced in between and the maximum difference defines the size requirement for the array. It is calculated based on Pressburger formulas using the OMEGA calculator [12]. De Greef et al. [4] and Darte et al. [3] extend this approach with linearization of multi-dimensional arrays. Zhao and Malik [15] present a method that measures the number of simultaneous alive variables in each iteration. This counting of live array elements is done by set operations (union, intersection) which are the whole or parts of the iteration domains. Grun et al. use the data dependency relations between the array references in the code to find bounds on the number of array elements produced or consumed by each assignment [7]. Then, a memory trace as a function of time is found. The peak memory trace contained within the bounding rectangles yields the total memory footprint. If the difference of boundaries for the critical rectangle is too large, the corresponding loop is split and the estimation is rerun in order to improve the estimation accuracy. In the worst case, a full loop unrolling is required to achieve a satisfactory estimate, which is unaffordable. Zhu et al. [16] propose to first decompose the array references into disjoint linearly bounded lattices. Then the memory size at the boundaries between the blocks of code is calculated. The maximum memory size inside each block is further estimated and the maximum found defines the overall memory footprint. Unfortunately, all the above listed techniques are still too computationally expensive to be performed frequently during system level design exploration. This is especially non-trivial when the applications become realistic and hence large.

978-3-9810801-2-4/DATE07 © 2007 EDAA

Balasa et al. [1] do not assume a fixed execution ordering. Data dependency analysis is performed and the total memory footprint is found through a greedy traversal of the relative graph. In order to deal with multiple read and write statements for the same array, arrays are partitioned into non-overlapping basic sets. During data flow graph traversal, each basic set is treated as a single unit with a fixed size. Basic set partitioning is however time consuming.

Kjeldsberg et al. [8] developed a fast technique which allows a partially fixed execution ordering. They first estimate the size of individual arrays by counting the number of nodes in the iteration domain at where array is written, constrained by the Extreme Dependency Vector (EDV). The EDV is the maximal projection of all dependency vectors (DVs), also called dependency distance vectors, on each iteration dimension, indicating the maximal lifetime window of all the array elements in each dimension. Then an inter-array size estimation is performed between arrays in such a way that arrays with non-overlapping lifetime can reuse the same memory locations. Their approach results in upper and lower bounds when the execution ordering is partially fixed. Their approach by itself is very fast, assuming the EDV is already given. However, calculating the EDV is still time consuming with the existing techniques like the OMEGA calculator [12] and Polylib library [10].

This paper contains two main contributions: array size estimation based on the Maximal Dependency Vector (MDV), and a fast calculation of the MDV. First, Section 2 explains how to estimate the array size based on the MDV. The MDV is the maximal DV among all DVs for the given execution ordering. As far as we know, we are the first using the MDV for performing the memory footprint estimation. Compared to previous work, the MDV based approach is very fast. The estimation itself is performed in the similar way as the EDV-based approach [8], but our MDV based approach is more accurate because it takes into account the execution ordering. Section 3 presents the second main contribution: two approaches for calculating the MDV (and EDV). The first is a general approach based on an ILP formulation. In order to perform a fast estimation, we also present an alternative called the vertexes approach. This can be used to calculate the MDV when the iteration domain is simplified to a bounding box, as it is in our case. Experiments on multiple test vehicles (Section 4) show that the estimation based on the vertexes approach MDV calculation is accurate when compared to the result achieved with the technique used in [4], while it is two orders of magnitude faster than [4]'s approach and the ILP approach MDV calculation. Finally, conclusions are drawn in Section 5.

2 Array Size Estimation

Our array size requirement estimation is based on a data dependency analysis between all array writes and reads. This analysis is performed using a geometrical model of the data and their accessing. We will use Fig. 1 to illustrate this and other concepts in this section focusing first on a single data write and read pair. The three loops in the code result in a three dimensional iteration space of iteration nodes, as depicted in Fig. 2(b). At each node, the statements within the loop nest are executed once, unless restricted by if-clauses. The set of iteration nodes at which a given statement is executed constitutes its iteration domain. We will give a more formal definition of this below. An array element is alive from the time it is written and until the time it is read for the last time. This lifetime can also be depicted as a lifetime window in the iteration domain. Since the array index expressions are restricted to be affine function of surrounding loop iterators in our targeted code, all array elements written within the lifetime window of one element are alive simultaneously. The maximal lifetime window among all lifetime windows constrains the maximal number of simultaneously alive array elements for the array. It hence defines the size requirement for the linearized array. For the given execution ordering, the maximum lifetime window is defined by the MDV. We will now illustrate how to estimate the memory footprint requirement using these principles.

```
for (x=0; x<=4; x++)
  for (y=0; y<=7; y++)
    for (z=0; z<=2; z++)  {
      A[x][y][z] = ...;
      if (x>=z) && (y>=x-z)
        ... = f(A[x-z][y-x+z][z]);
  }
```

Figure 1: Code example

For this example, array A is written in one statement and read in another statement. Assuming it is not used anywhere else, the array read consumes the data elements. x, y and z refer to the values of the loop variables at the time the write is performed and x', y' and z' refer to the values of the loop variables at the time the read is performed. The loop variables are used to define iteration domains, denoted as I and I' respectively, within which the array is written and read.

$$I = \{[x,y,z] | 0 \le x \le 4 \wedge 0 \le y \le 7 \wedge 0 \le z \le 2\}$$
$$I' = \{[x',y',z'] | 0 \le x' \le 4 \wedge 0 \le y' \le 7 \wedge 0 \le z' \le 2 \wedge x' >= z' \wedge y' >= x' - z'\}$$

In our case, all polyhedron representations of iteration domains are simplified to bounding boxes. With this bounding box simplification, the counting of the lattice points of a polytope is much simpler and faster than for a general polytope. The bounding box approach, as also used in [8], still gives reasonable result as the data accesses for our targeted applications are quite regular. A bounding box iteration domain for one statement is in general expressed as

$$I = \{\vec{i} = [i_1, \ldots, i_m]^T \mid \bigwedge_{g=1}^{m} L_g \le i_g \le U_g\} \qquad (1)$$

in which L_g and U_g means the lower bound and the upper bound values at the gth loop dimension. In the above ex-

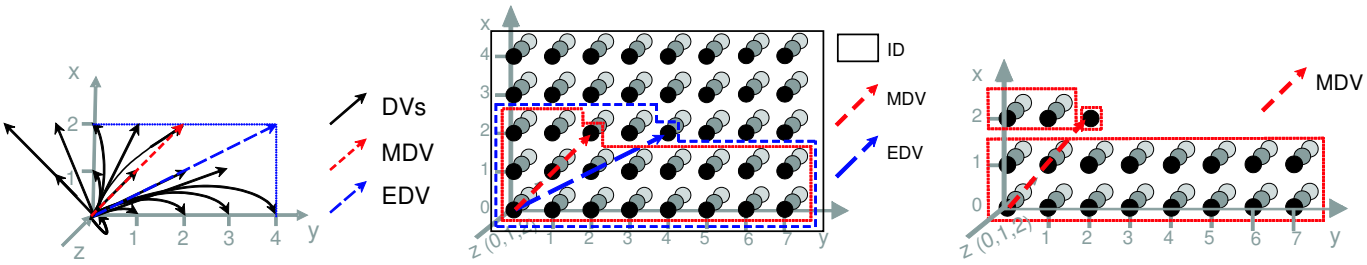

Figure 2: (a) DVs, EDV and MDV and (b) maximal lifetime constrained by EDV/MDV and (c) size requirement counting on MDV

ample, the bounding box expression for the write statement exactly represents the original polytope shape given earlier, while the bounding box of the read statement is an approximated of the original given by

$$I' = \{[x', y', z'] | 0 \leq x' \leq 4 \wedge 0 \leq y' \leq 7 \wedge 0 \leq z' \leq 2\}$$

Fig. 2(a) illustrates all the DVs between the array writes and reads in our code example. In Fig. 2(a) some DVs are negative in the y-dimension. In this case they appear because of the bounding box simplification, but for other examples they may appear in the original code. As long as they are valid with the given execution ordering, they will in any case be taken into account. As mentioned, the array size requirement is equal to the maximum lifetime window constrained by the MDV. The MDV will always be one of the existing DVs. The MDV must be one of the DVs with the largest dependency distance at the outermost dimension. Among these candidates, it must be one of the DVs with the largest dependency distance at the second outermost dimension. This continues until there is just one DV at one inner dimension. For our example, the MDV is $[2,2,0]$ w.r.t. the x-, y- and z-loop dimensions and it is always larger than other DVs, e.g., $[2,1,0]$, $[1,3,0]$, $[2,-2,0]$ and $[0,4,0]$ as shown in Fig. 2(a).

As mentioned, the MDV defines the maximal lifetime window among all array element accesses. When there is a one-to-one mapping between the data elements and iteration nodes, i.e., each iteration node accesses one data element, the size requirement can be calculated by counting the number of iteration nodes within the array write iteration domain constrained by the MDV. If there is not a one-to-one mapping, e.g. data are accessed at every N iterations, the end size requirement should be divided by the interval N. This can easily be handled by postprocessing. As the array index expressions are affine functions of the surrounding loop dimensions and the iteration domains are represented as bounding boxes, the counting procedure will start from the first iteration node at where the array is written and continue to the node reached by the MDV as shown in Fig. 2(b). The counting is formulated as

$$size = \sum_{g=1}^{m-1} \left(MDV[g] \cdot \prod_{l=g+1}^{m} (U_l - L_l + 1) \right) + MDV[m] + 1 \quad (2)$$

in which m is the number of loop dimensions of the array write iteration domain. In the formula, the counting is summed over each loop dimension starting from the outermost dimension. Let us illustrate it on Fig. 2(c). The MDV distance of the outermost x-dimension is first multiplied with the bound difference, $U_l - L_l + 1$, of all inner dimensions. The resulting number of iteration nodes is 48 as the distance is 2 and the bound ranges of y and z are 8 and 3 respectively. This procedure is repeated at the y dimension where the number of nodes is 6. The innermost dimension, z, is treated separately by the second part of the formula, since it is not going to be multiplied with anything. In this case, the MDV distance of the z-dimension is zero. The last constant one is added in order to count the iteration node reached by the MDV. The size requirement for this example is hence 55.

The EDV used in [8] is a somewhat different concept than the MDV used here, but with many similarities. It is found using the maximal dependency distance among all the DVs at each loop dimension. This difference stems from the fact that this technique does not take into account the application's (full) execution ordering. For our example, the EDV is equal to $[2,4,0]$, which is in fact not one of the original DVs. When using the EDV to calculate the storage size requirement for the given execution ordering, we can count the iteration nodes constrained by the EDV as shown in Fig. 2(b). The size requirement turns out to be 61, which results in overestimation compared to the MDV based method. However, it is still necessary to use the EDV in [8] since the execution ordering in their cases can be partially fixed or even unfixed. When the execution is partially fixed, their approach will give lower bound and upper bound estimations. For a fully fixed ordering, the MDV based approach would result in a more accurate array size estimation when the EDV is different from the MDV.

In cases where the distance of the MDV/EDV at any loop dimension has a value negative or larger than the distance difference of the upper bound and lower bound of this dimension, Eq. 2 cannot count the number of iteration nodes constrained by the MDV correctly. This is easy to deal with, and is included in our prototype tool implementation. Due to page limitations we will not discuss it further.

The approximation to bounding boxes usually works very well in practice. Iteration domains are typically very close to rectangular. A major exception are skewed loops and trian-

gular loops: for these, the dependency size may be overestimated with a factor of two. Typically, however, only a few loop dimensions are not rectangular. It is then still possible to use the MDV approach combined with exact counting of the number of points in the iteration domain. In Eq. 2, the non-rectangular factors of the product term should then be replaced by the exact counting function. Note that the use of MDV instead of the exact dependency relation still leaves a degree of inaccuracy, but we believe this is negligible for practical purposes.

Above we have discussed how to estimate the array size requirement based on the EDV/MDV when there is a dependency between one statement writing to an array and one statement reading from the same array. In general, multiple writes and reads of the same array may exist. Every write-read pair may lead to a data dependency, if they access the same elements. If two write-read pairs access non-overlapping data, their sizes can be computed individually and summed. If on the other hand they overlap, the overlapping data needs to be stored only once. For the overlapping part, only the maximum of the two sizes needs to be stored. Basic set analysis [1] splits dependencies in this way. It needs to be performed only once, since behavior-preserving transformations cannot change the dependency relations. In our implementation, however, we do not go to basic set analysis but simply take the maximum of the two sizes if the dependencies are overlapping.

3 MDV/EDV Calculation

In this section we discuss the other main contribution of this paper: how to perform a fast calculation of the MDV. First a general ILP approach is presented. This is followed by a fast vertexes approach, which is a simplification of the general ILP approach in such a way that we do not actually need to solve the ILP problems. Both approaches also work for the EDV calculation. For each approach, we will first present how to calculate the EDV and then the MDV, as calculating the MDV is the same as the EDV calculation but with extra constraints added.

3.1 ILP approach

We now present the formal ILP formulation of calculating the MDV/EDV for dependencies between each pair of array write and read. Their iteration domains, denoted I and I', consist of a set of inequality constraints. When data flow dependencies exist, the index expressions of array write and read, denoted E and E' respectively, must be equal at each of the array dimensions. This is because both write and read should access the same data elements. This results in a set of equalities:

$$E'_k = E_k \mid 1 \leq k \leq n \tag{3}$$

in which n is the number of array dimensions. Note that in this example, n is the same as the number of loop dimensions m, but it is not necessarily the case in general.

For our example, the set of equalities are $x' - z' = x$, $y' - x' + z' = y$ and $z' = z$ for the three array dimensions.

To find the EDV distance at one loop dimension it is necessary to calculating the maximal dependency distance projection at that dimension among all the DVs. This can be achieved by solving for each loop dimension the ILP problem:

$$MAX(i'_g - i_g) \tag{4}$$

based on the set of inequality constraints for I and I' and the set of equality constraints expressed in Eq. 3. In the equation, g is the analyzing dimension of the array write iteration domain. For the above example, we need to solve the ILP maximal problems at each loop dimension, i.e., $MAX(x' - x)$, $MAX(y' - y)$, $MAX(z' - z)$. The EDV is hence equal to $[2,4,0]$. Note for this simple example, the ILP problem is easy to solve. For the more general real-life applications, there are usually more loop dimensions, and arrays can also have more dimensions and/or more complex index expressions. The complexity of the ILP problem will then grow exponentially.

The MDV is similarly calculated by solving the ILP maximal problems at each loop dimension sequentially, starting at the outermost dimension. In addition, the calculated maximum distances of the outer dimensions are propagated as equality constraints to the calculation of the dependency distance at the inner loop dimensions. The reason for this is that we want to calculate the actual dependency distance at the analyzing dimension for the MDV, which is not necessarily the projected maximal dependency distance at this dimension. The dependence distances calculated at the outer dimensions for the MDV with their equality constraint expressions are therefore required to be taken into account. For the above example, we first solve the ILP problem $MAX(x' - x)$ giving a dependency distance at the outmost dimension equal to 2. By propagating the equality constraint $x' - x = 2$, we then solve the ILP problem $MAX(y' - y)$ and find the dependency distance at the second outermost dimension to be 2. We now propagate both outer equality constraints and solve the ILP problem $MAX(z' - z)$. This gives a dependency distance at the innermost dimension equal to 0. From this we get the MDV equal to $[2,2,0]$. With the ILP approach, the EDV/MDV can be calculated using either the original iteration domain or the bounding box iteration domain representation.

3.2 The vertexes approach

Above we have shown how to calculate the MDV/EDV by solving an ILP problem at each loop dimension. Since solving ILP problems is computationally expensive, we propose a simplified approach, named vertexes approach. We exploit the special form of the ILP problem and the bounding box approximation to directly calculate the MDV/EDV values without having to solve the ILP problems.

We can partly solve the ILP problem by rewriting one of

the index equalities of Eq. 3:

$$e_g^{h'} \cdot i'_g + O' = e_g^h \cdot i_g + O \tag{5}$$

in which e_g^h and $e_g^{h'}$ are the index coefficients of the analyzed loop dimension g for array write and read, at the analyzed array dimension h. O and O' are the parts of the index expressions that do not contain i_g or i'_g. We can rewrite this formula to isolate the difference between i'_g and i_g which we want to maximize:

$$i'_g - i_g = \frac{(e_g^h - e_g^{h'}) \cdot i_g + O - O'}{e_g^{h'}} \tag{6}$$

If $e_g^{h'}$ is equal to zero, we have to use e_g^h instead. The MAX problem in Eq. 4 is hence converted to

$$MAX(i'_g - i_g) = \begin{cases} MAX(\frac{(e_g^h - e_g^{h'}) \cdot i_g + O - O'}{e_g^{h'}}) & \text{when} \quad e_g^{h'} \neq 0 \\ MAX(\frac{e_g^h \cdot i'_g + O - O'}{e_g^h}) & \text{when} \quad e_g^{h'} = 0 \end{cases} \tag{7}$$

If $|e_g^{h'}| \neq 1$, Eq. 7 is only an approximation, because the constraint that all variables are integral is ignored. This, however, only makes a difference for border cases of an array, which contribute little to the total size. This does not occur much in practice. If we make the real division and round the result up, we find a maximum which may be an overestimate. This overestimation due to the border cases can be removed if basic set analysis [1] is applied before our estimation is performed.

The right-hand side of Eq. 7 is a linear combination of loop variables. Because bounding box constraints are assumed on the loop variables, we can easily find the maximum of this right-hand side by replacing a loop variable with its upper bound if it has a positive coefficient, and its lower bound if it has a negative coefficient. In other words, the maximum is found at one of the vertexes of the bounding box, and we can immediately find which vertex by looking at the coefficients of Eq. 7.

If the array has more than one dimension, we can find a formulation of Eq. 7 for each array dimension. We then have to find a solution to the ILP which satisfies all array dimensions simultaneously. The maximum difference will certainly be smaller than or equal to the minimum of all maxima found with Eq. 7. The minimum among the maxima is the dependency distance for the EDV at that dimension.

For the example of Fig. 1, the vertexes approach is applied as follows. At the outermost loop dimension x, we calculate the maximal value at the first array dimension with the equality $x' - z' = x$. Based on Eq. 6, which is transformed as $x' - x = z'$. This leads to the ILP problem $MAX(x' - x)$, which is equivalent to solving the problem $MAX(z')$ based on Eq. 7. By taking the upper bound value of z', the maximal value for the first array dimension at the outermost loop dimension is equal to 2. Similarly, the maximal value is 9 at the second array dimension. The third dimension does not constrain x or x', so it does not affect the maximum. The minimal maximum among all array dimensions is the dependency distance at the outermost loop dimension, that is 2. In a similar way, the dependency distance for EDV at other loop dimensions can be calculated. For the MDV calculation, the variables which contribute to the actual dependency distance calculation at one loop dimension are propagated to the dependency distance calculation at the inner loop dimensions using their fixed values.

When the same array element is written multiple times, calculation of EDV/MDV based on write-read pairs is not fully accurate: it includes the time between the reading of an element and its overwriting into the dependency. Preprocessing the index expressions into dynamic single assignment (DSA) form [13] avoids this problem. It has to be performed only once so that is a quite acceptable overhead even in a system exploration context. Indeed, once the initial code is in DSA form, also the code transformations become simpler and they will maintain the DSA form.

4 Experiments

We have developed a prototype tool in the script language *Python* for the memory footprint estimation and also the two approaches of the MDV calculation. For the ILP approach based MDV calculation, the problems are created in Python and then solved by calling the ILP solver. In this case, GNU Linear Programming Kit (GLPK) [6] is called which is written in language *C*. We have performed experiments on several real life test vehicles and compare our results with what Atomium/MC achieves. Atomium/MC is implemented based on the techniques presented in [4]. In the current version, the memory footprint for one application is simply the sum of the size requirements for all arrays. To make a fair comparison, the size requirements for all arrays calculated by Atomium/MC is also summed up. Their approach is different from ours in that they calculate the maximal distance at the data domain by solving ILP problems after linearization of the array indices. The experiments are performed on a workstation with Intel Pentium 4 2.4GHz processor and 1G memory.

Application code		declared size	MDV approach size	MDV approach $t_{vertexes}$	MDV approach t_{ILP}	Atomium/MC size	Atomium/MC t_{MC}
Cavity Detection	orig.c	2536880	1016984	124.0ms	5.8s	1016984	24.9s
	lt_1.c	2536880	760987	114.3ms	6.0s	760987	27.1s
	lt_2.c	2536880	5743	166.4ms	6.0s	5743	26.5s
	lt_3.c	2536880	3838	168.5ms	6.0s	3838	82.4s
QSDPCM	orig.c	118906	117338	0.7s	14.9s	117338	73.1s
	lt.c	118906	85128	1.1s	12.4s	85128	144.0s

Table 1: Estimation comparison for Cavity Detection & QSDPCM

The first application is the Cavity Detection Algorithm used for detection of cavity perimeters in medical imaging. The second is the QSDPCM algorithm, which is an interframe compression technique for video images, involving hierarchical motion estimation and a quadtree-based encoding

of the motion compensated frame-to-frame differences. Our estimation is performed on a number of code versions resulting from different loop transformations. In the Tab. 1, t means the CPU execution time. As shown, our memory footprint estimates are identical to the optimized results reached by Atomium/MC. When comparing the computation time, our vertexes approach based estimation takes less than 200ms for all versions of Cavity Detection codes. For QSDPCM, it takes approximately one second. Our ILP based estimation is in general one order of magnitude slower, while Atomium/MC is two orders of magnitude slower than our vertexes approach.

Application code	Declared size	MDV approach			Atomium/MC	
		size	$t_{vertexes}$	t_{ILP}	size	t_{MC}
durbin.c (N=500)	502000	251498	135.0ms	6.2s	251003	56.8s
dynprog.c	1020001	19701	137.7ms	10.9s	19604	17.0s
gauss.c	5280008	1436016	25.5ms	4.4s	1436008	18.3s
reg_detect.c	8392	4337	26.5ms	4.5s	4297	12.2s

Table 2: Estimation comparison for other applications

The memory footprint estimation is also performed for several other applications. Our estimation is also very accurate compared to what is achieved with Atomium/MC. When comparing the execution time among different approaches, the computational time of our vertexes approach is on average two orders of magnitude faster than our ILP approach and Atomium/MC respectively. Remember that our prototype tool is implemented in the script language Python while Atomium/MC is implemented in C++. Our algorithm will be even faster when implemented in C/C++.

A comparative evaluation in terms of speed and accuracy with other fast estimation techniques presented in Section 1 would be interesting. This is hard to do in the absence of their benchmark algorithms. In general, both the MDV calculation presented in Eq. 7 and the memory footprint estimation presented in Eq. 2 have very low complexity. This is not the case for the other techniques as motivated in section 1. Our approach should therefore still compare very favorably.

5 Conclusions

This paper presents a system level memory footprint estimation technique based on the MDV calculation. Our MDV based estimation produces accurate results as shown on multiple test-vehicles. We have also presented a new general ILP approach and a novel vertexes approach for MDV calculation. Our vertexes approach is extremely fast compared to other approaches. This makes our estimation approach especially useful during system level exploration, where estimates must be frequently repeated. Our MDV calculation methods can also be used for fast EDV calculation to save computation time for existing estimation techniques. For future work, we will focus on estimation when more complex execution ordering occurs between array writes and reads. Other important issue is to consider in-place mapping between multiple arrays and also to extend the estimation for multi-processors.

References

[1] F. Balasa, F. Catthoor, and H. De Man. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on VLSI Systems*, 3(2):157–172, June 1995.

[2] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publ., Boston, MA, 2002. ISBN 0-7923-7689-7.

[3] A. Darte and Y. Robert. Lattice-based memory allocation. *IEEE Transaction on Compuaters*, 54(10):1242–1257, Oct. 2005.

[4] E. De Greef, F. Catthoor, and H. De Man. Array placement for storage size reduction in embedded multimedia systems. In *Proc. Int. Conf. on Applic.-Spec. Array Processors*, pages 66–75, Zurich, Switzerland, July 1997.

[5] C. H. Gebotys and M. I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *Proc. ACM/IEEE Design Automation Conf.*, San Jose CA, USA, Nov. 1991.

[6] GLPK: GNU Linear Programming Kit, `http://www.gnu.org/software/glpk/`.

[7] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (CODES)*, pages 145–149, Seattle, WA, Mar. 1998.

[8] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. Data dependency size estimation for use in memory optimization. *IEEE Trans. on Comp. Aided Design*, 22(7):908–921, July 2003.

[9] F. Kurdahi and A. Parker. Real: a program for register allocation. In *Proc. ACM/IEEE Design Automation Conf.*, Miami FL, USA, June 1987.

[10] V. Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report.

[11] S. Y. Ohm, F. J. Kurdahi, and N. Dutt. Comprehensive lower bound estimation from behavioral description. In *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, USA, Nov. 1994.

[12] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, Aug. 1991.

[13] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. In *Proc. 3rd Asian Symp. on Programming Languages and Systems (ASPLAS)*, volume 3780 of *Lecture Notes on Comp. Sc.*, pages 330–346, Tsukuba, Japan, Nov. 2005.

[14] I. Verbauwhede, C. Scheers, and J. M. Rabaey. Memory estimation for high-level synthesis. In *Proc. 31st ACM/IEEE Design Automation Conf.*, pages 143–148, San Diego, CA, June 1994.

[15] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *Proc. 36th ACM/IEEE Design Automation Conf.*, pages 811–816, New Orleans, LA, June 1999.

[16] H. Zhu, I. I. Luican, and F. Balasa. Exact computation of storage requirements for multi- domensional signal processing applications. In *11th Proc. IEEE Asia and South Pacific Design Autom. Conf. (ASPDAC)*, Yokohamma, Japan, Jan. 2006.