Exact Memory Size Estimation for Array Computations without Loop Unrolling

Ying Zhao and Sharad Malik
Department of Electrical Engineering
Princeton University
Princeton, New Jersey

Email: {yingzhao, sharad}@ee.princeton.edu

Abstract

This paper presents a new algorithm for exact estimation of the minimum memory size required by programs dealing with array computations. Memory size is an important factor affecting area and power cost of memory units. For programs dealing mostly with array computations, memory cost is a dominant factor in the overall system cost. Thus, exact estimation of memory size required by a program is necessary to provide quantitative information for making high-level design decisions.

Based on formulated live variables analysis, our algorithm transforms the minimum memory size estimation into an equivalent problem: integer point counting for intersection/union of mappings of parameterized polytopes. Then, a heuristics was proposed to solve the counting problem. Experimental results show that the algorithm achieves the exactness traditionally associated with totally-unrolling loops while exploiting the reduced computation complexity by preserving original loop structure.

1 Introduction

Due to the fast increase in size and complexity of IC systems, high-level design and power optimization techniques become two very important research topics. To make proper high-level design decisions, such as algorithm selection, hardware-software partition, trade-off between various optimization techniques, we need techniques that can exactly and quantitatively measure certain cost functions, which reflect area, speed and power consumption of the IC systems. For programs dealing mostly with array computations, such as applications in DSP and video signal processing domain, due to the large amount of data and computations being involved, power consumption of memory accesses and storages is responsible for a large proportion of the power cost of the whole system. The power cost of each memory operation will increase as the size of memory unit increases, which makes the memory size one of the dominant factor affecting the system power cost. Thus exact estimation of memory size required by a program is a necessity in high-level design field.

DAC 99, New Orleans, Louisiana

The minimum memory size is equal to the maximum number of live variables at any time during the program execution. For programs dealing only with scalars, the estimation of minimum memory size when the schedule is fixed is relatively straightforward, which involves counting the number of live variables after execution of each instruction. Since the number of variables and instructions involved is rather limited, the life time of each variable can be analyzed individually. The minimum memory size can be calculated with reasonable computation complexity. Even when the schedule is not fixed, the problem of finding the schedule with least number of memory locations is well-formulated. Although it is still NP-hard, some heuristics have been introduced [6], which achieve satisfactory results.

However, the main bodies of DSP and video signal processing programs are loops and their data objects are mostly multi-dimensional arrays. Due to formidable instruction and data size, it is unrealizable to unroll all loops in the programs, treat each array element as a scalar and count the number of live variables after execution of each instruction. So it is impossible to extend the results for scalars directly. Some new methods that deal with loops and arrays specifically have to be developed.

Unlike other cost functions, such as execution time, which have very straightforward forms, memory size can not be represented explicitly. Thus, memory size estimation for programs dealing with arrays has not been widely studied until now. Researchers from IMEC proposed several methods to solve this problem [5, 4]. Their methods achieve good trade-off between exactness of the estimation and computational complexity. However, to get the absolute lower bound of memory size, their methods may require all loops in the program to be unrolled in the worst case. We will discuss their work in detail in section 7.

A new algorithm to exactly estimate the minimum memory size for programs dealing with arrays is proposed in this paper. Based on formulized live variable analysis, it transforms the memory size estimation into an equivalent mathematical problem which can be solved by integer point counting for intersection/union of mappings of polytopes. Experimental results on some typical DSP applications show that the algorithm gives the exact estimation of minimum memory size without unrolling loops.

The rest of this paper is organized as follows. In section 2, the polytope model for perfectly nested loop is described. Section 3 gives a brief review of the integer point counting problem. Section 4 proposes our algorithm to estimate the minimum memory size required by a perfectly nested loop at both iteration level and statement level and then extends

to deal with imperfectly nested loops and whole programs. Section 5 shows the heuristic for solving counting problems. Experimental results are presented in section 6 to show the exactness of our algorithm. Section 7 introduces some related work.

2 Problem Overview

2.1 Polytope Model

Since the main bodies of DSP progrms are loops, we start with single perfectly nested loops. As can be seen later, the algorithm working on perfectly nested loops can be extended to deal with imperfectly nested loops and whole programs.

Based on the polytope model [7], a perfectly nested n-deep loop is defined in the iteration space IS, which is a polytope in space Z^n bounded by the loop bounds. Every point I in IS is an n-tuple that corresponds to a particular iteration.

In this model, a loop can be described as:

$$\begin{aligned} I \in IS, \quad & S_1 : Y_1[f_1(I)] = F_1(X_{11}[g_{11}(I)], X_{12}[g_{12}(I)] \ldots) \\ & \vdots \\ & S_i : Y_i[f_i(I)] = F_i(X_{i1}[g_{i1}(I)], X_{i2}[g_{i2}(I)] \ldots) \\ & \vdots \end{aligned} \tag{1}$$

S_i is the ith statement in the loop nest;
$Y_i[f_i(I)]$ is an array defined by S_i;
F_i is the function performed by S_i;
$X_{ij}[g_{ij}(I)]$s are source operands of F_i;
$f_i(I)$ and $g_{ij}(I)$ are index functions. They are mappings from $Z^n \rightarrow Z^m$ where m is the dimension of the corresponding array.

In most cases, the index functions are affine functions of loop indices. In other words, $f_i(I)$ and $g_{ij}(I)$ can be represented as $AI + b$, where A is a $m \times n$ matrix (index matrix), b is a $m \times 1$ constant vector.

2.2 Minimum Memory Requirement

Given a loop as described in (1) and assuming primary inputs(PIs) are stored in the memory before the execution of the loop and primary outputs(POs) will be stored in memory after the execution, we can claim:

$$Min\_size = max(\#(PIs), \#(POs), Max_t(\#(live\_vars_t))$$

Min_Size: number of memory locations required;
t: any time instant during the loop execution.

It is obvious that estimation of minimum memory size involves exact counting of the number of live variables at every time instant during the loop execution.

2.3 A Simple Example

We use a loop for matrix multiplication to illustrate the basic idea of minimum memory size estimation.

```
for i= 1 to n do
  for j= 1 to n do
    for k=1 to n do
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

Suppose matrices A, B are PIs and C is PO. Before the execution, $2n^2$ memory locations are required to store A and B. When $C[1,j](1 \leq j \leq q-1)$ are computed, all the elements of A and B have to be kept alive, since they will be used later. New memory locations have to be allocated to store the elements of C. After $C[1,n] = C[1,n] + A[1,1] * B[1,n]$ has been executed, $A[1,1]$ will not be used again. So $C[1,n]$ can use the location of $A[1,1]$. From now on, no new memory locations have to be allocated. The minimum memory size required is: $2n^2 + n - 1$. A complete description of memory allocation for the above example is shown in Figure 1.

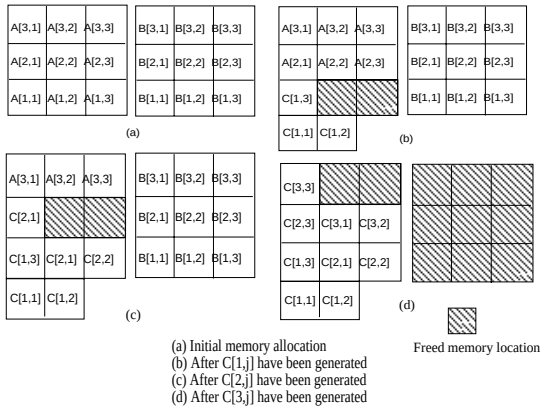


Figure 1: Memory allocation during 3X3 matrix multiplication

3 Integer Point Counting

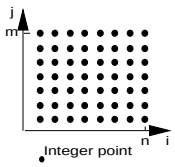
Our algorithm transforms the estimation of minimum memory size into an equivalent mathematical problem: integer point counting of intersection/union of mappings of parameterized polytopes. Figure 2 shows the basic problems for integer point counting and their relationship with array element access. Polytopes are used to represent the iteration space of nested loops and mappings of polytopes correspond to array elements accessed by the loop. The polytopes in our algorithm are with parameterized bounds. While integer point counting for images of parameterized polytopes is still a problem under research [2, 3, 8], based on the characteristics of array access pattern in most DSP programs, we propose a heuristic to deal with it (section 5).

4 Estimate Minimum Memory Size

Although lots of work has been done for memory size estimation for scalar programs, for programs dealing with arrays, it is unrealistic to unroll all the loops and treat each array element as a scalar due to the intractable increase of problem size. Besides, since the number of iterations may be very large, it is prohibitively expensive to check the number of live variables after the execution of each instruction. Fortunately, the execution order of iterations in a loop can be described formally and the index functions of arrays are mostly affine functions of loop indices. As shown later, through the algorithm we proposed, a uniform function to represent the number of live variables after execution of each iteration can be derived when the schedule is fixed. Thus, the minimum memory size can be computed by finding the maximum value of the function over all iterations.

1. Integer point counting for parameterized polytopes

Polytopes P1={(i,j)|1<=i<=n, 1<=j<=m}
The target is to count the number of integer points in P1.

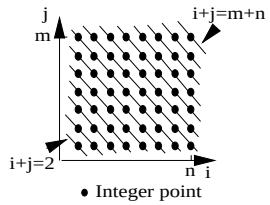


Loop: for i = 1 to n do
for j = 1 to m do
...

Integer point counting gives out the number of iterations in the loop.

2. Integer point counting for mappings of parameterized polytopes

Map polytopes P1 along vector(1,−1)
The target is to count the number of distinctive values of i+j when (i,j) is in P1.



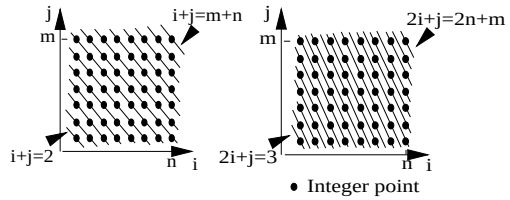
Loop: for i = 1 to n do
for j = 1 to m do
A[i+j] = ...

Integer point counting gives out the number of elements of A accessed by the loop.

3. Integer point counting for intersection/union of smappings of parameterized polytopes

Map P1 along vector (1,−1) and (2,−1) and enumerate the intersection of the two mappings.

The target is to count the number of integers a, such as a satisfies both of the following two conditions:
(1) there exist (i1,j1) in P1, such that i1+j1 = a ;
(2) there exist(i2,j2) in P2, such that 2i2+j2 = a .



Loop1: for i =1 to n do
for j =1 to m do
A[i+j] = ...

Loop2: for i = 1 to n do
for j = 1 to m do
.... = A[2i+j]

Integer point counting gives out the number of elements of A that are defined in Loop1 and used in Loop2.

Figure 2: Integer point counting vs. array access

In this section, we consider memory size estimation for perfectly nested loops. We assume that all live variables are stored in memory, since the size of register files is trivial compared to the memory size.

4.1 Statement Level Estimation

Suppose there are N statements in a perfectly nested loop, $M_i(I^*)$ is the number of live variables after the ith statement S_i has been executed at iteration I^*. The memory size required can be represented as:

$$Min\_Size = max_i(max_{I^*}(M_i(I^*) \mid I^* \in IS)) \quad (2)$$

As can be seen in section 4.2, we can get each $M_i(I^*)$ as a function of I^*. Basically, we count the number of live variables after execution of each statement, thus we call this method statement level estimation. Although we can get exact answer through this method, the computation complexity is high, both for deriving multiple functions and computing maximum value for them. Theorem 1 enables us to simplify the computation.

Theorem 1 *For a loop with N statements,*
$| Min\_Size - max_I(M_j(I) \mid I \in IS) | \leq \triangle, 1 \leq i \leq N.$

Min_Size is minimum memory size required;
$\triangle = max(N-1, \mid N-1-\#(S_1)-\ldots-\#(S_N) \mid)$;
$\#(S_i)$ is number of source operands in S_i.
$\triangle$ is a number comparable to the number of statements in the loop. It is very small comparing to the overall memory size. That means, $max_I(M_i(I) \mid I \in IS)(1 \leq i \leq N)$ are all very close to the minimum memory size. So it is enough to compute any $max_I(M_i(I) \mid I \in IS)$ and use it to approximate the Min_Size. In section 4.2, we will show the algorithm to compute $max_I(M_N(I) \mid I \in IS)$, which is called iteration level estimation, since the number of live variables is counted after the execution of each iteration.

4.2 Iteration Level Estimation

We illustrate the iteration level estimation process with the following example, where matrices A, B are PIs and D are POs:

```
for i=1 to 8 do
  for j=1 to 7 do
    C[i,j] = A[i,j] + B[i,j];
    C[i+n,j] = A[i,j] * B[i,j];
    D[i,j] = C[2i,j] + C[i+j,j];
```

After the execution of iteration I^*, IS is divided into:

Executed iterations $P1(I^*) = \{I \mid I \preceq I^*\}$
Unexecuted iterations $P2(I^*) = \{I \mid I \succ I^*\}$

Figure 3 shows an example for the partition.

The variables that satisfy both the following two conditions should be kept alive:

a. they have been generated before:

$$S1 = \{(PIs) \cup (\cup(Y_i[f_i(I)] \mid I \in P1(I^*)))\} \quad (3)$$

where: $(\cup(Y_i[f_i(I)] \mid I \in P1(I^*)))$ covers all the destination operands of the executed iterations. It includes all the variables that have been defined by the loop.

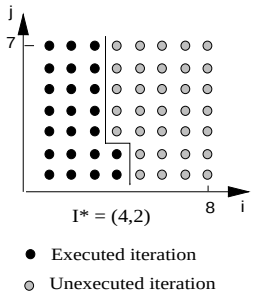


Figure 3: Example of iteration space partition

b. they will be accessed later:

$$S2 = \{(POs) \cup (\cup(X_{ij}[g_{ij}(I)]) \mid I \in P2(I^*))\} \quad (4)$$

where: $(\cup(X_{ij}[g_{ij}(I)] \mid I \in P2(I^*)))$ covers all the source operands of the unexecuted iterations. It includes all the variables that will be accessed by the loop later.

For the above example,

$$S1 = A \cup B \cup (C[i,j] \cup C[i+n,j] \cup D[i,j] \mid (i,j) \in P1)$$
$$S2 = D \cup (A[i,j] \cup B[i,j] \cup C[i+j,j] \cup C[2i,j]) \mid (i,j) \in P2)$$

The number of variables in the intersection of $S1$ and $S2$ equals the memory size required at current time instant. In the actual computation process, we first count the number of live variables in each array individually. For array C in the above example, to get the number of its live elements just after execution of iteration I^*. Two sets of elements have to be computed:

1. The elements of C that have been generated. Since the elements of C can be defined by either one of the first two statements, the set of (i,j) pairs and $(i+n,j)$ pairs that have been touched by the executed iterations cover the coordinates of all the generated elements, which can be represented as:

$$\{([i,j] \mid (i,j) \in P1\} \cup \{[i+n,j] \mid (i,j) \in P1\}$$

2. The elements of C to be accessed later. These are the elements of C whose coordinates equals one of $(i+j,j)$ pairs or $(2i,j)$ pairs that will be touched by the unexecuted iterations. Their coordinates can be represented as:

$$\{[i+j,j] \mid (i,j) \in P2\} \cup \{[2i,j] \mid (i,j) \in P2\}$$

The elements of C whose coordinates have been touched by the executed iterations and will also be touched by the unexecuted iterations have to be kept alive. Thus, by counting the number of different coordinates in the intersection of the above two sets, we get the number of live elements of C just after the execution of iteration I^*.

Since I^* is not fixed, P1 and P2 are parameterized polytopes. The basic operations involved are enumerating the intersection/union of mappings of parameterized polytopes. After the enumerations have been done, we get a symbolic function of the number of live variables in each array, with the I^* as free variable. Sum up the functions of all arrays, we get the overall minimum memory size required by the loop currently as a function of I^*. The remaining problem is to find the maximum value of this function over all iterations. The whole process of our algorithm is shown below.

```
Algorithm Mem_size(IS, Min_Size)
{
    Inputs:IS(iteration space)
    Output:Min_Size(minimum memory size)
    P1 = {I | I ≺ I*};
    P2 = {I | I ≻ I*};
    foreach array A
    {
      if A is PIs Im1(A) = A;
            /* Im1: Generated elements */
      else foreach access of A as destination
        Im1(A) = Im1(A) ∪ Map(P1, f_i);
            /* f_i : index function */
      if A is POs Im2(A) = A;
            /* Im2: Elements to be accessed */
      else foreach access of A as source
        Im2(A) = Im2(A) ∪ Map(P2, g_i);
            /* g_i : index function */
      Im(A) = Im1(A) ∩ Im2(A);
      Mem_size(I*) += Count(Im(A));
    }
    foreach iteration I* ∈ IS
    {
       compute Mem_size(I*);
       if Min_Size < Mem_size(I*);
         Min_Size = Mem_size(I*);
    }
}end algorithm
```

For further clarification of our method, we illustrate the algorithm using the matrix multiplication example.

Just after iteration $I^* = (i^*, j^*, k^*)$,
$P1 = \{(i,j,k) \mid (i,j,k) \preceq I^*\}$;
$P2 = \{(i,j,k) \mid (i,j,k) \succ I^*\}$;
For each array, we have:

$$Im1(A) = A, Im2(A) = Map(P2, space(i,k))$$
$$Im1(B) = B, Im2(B) = Map(P2, space(j,k))$$
$$Im1(C) = Map(P1, space(i,j)), Im2(C) = C$$

$Im1(\triangle)$ covers generated elements of array $\triangle$;
$Im2(\triangle)$ covers the elements to be accessed in array $\triangle$.
Number of live variables in each array are:

$$Count(Im(A)) = \begin{cases} (n-i^*+1)n & j^* < n \\ (n-i^*)n + (n-k^*) & j^* = n \end{cases}$$
$$Count(Im(B)) = \begin{cases} n^2 & i^* < n \\ (n-j^*)n + (n-k^*) & i^* = n \end{cases}$$
$$Count(Im(C)) = (i^*-1)n + j^*$$

where:

$Im(\triangle) = Im1(\triangle) \cap Im2(\triangle)$, which covers the coordinates of the live variables array $\triangle$;

Count($Im(\triangle)$) gives out the symbolic enumeration of $Im(\triangle)$, with i^*, j^*, k^* as free variables.

When $i^* < n, j^* = n, k^* = 1$, the minimum memory size is $2n^2 + (n-1)$. The result is the same as our analysis in section 2.3.

4.3 Imperfectly nested loop

Until now, we have solved memory size estimation for single perfectly nested loops. However, in real DSP and video signal processing applications, imperfectly nested loops are frequently encountered. Besides, a whole program can be

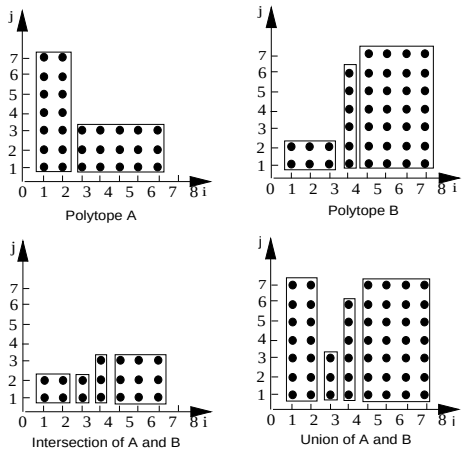


Figure 4: Intersection/union of polytopes

considered as an imperfectly nested loop, whose outermost loop has just one iteration.

In imperfectly nested loops, some loop nests are parallel with other loops nests or blocks of statements within some outer loop nests. Usually, there are multiple iteration spaces. Here, we only consider loops without conditional statements. However, our algorithm can be extended to deal with loops with conditional statements, since it can be applied to all the possible execution paths and the maximum number of live variables at any time over all the execution paths is the minimum memory size required.

Theorem 2 *For an imperfectly nested loop, we only need to do iteration level estimation for all the innermost loop nests to get the minimum memory size required.*

The algorithm proposed in section 4.2 can still be applied to do iteration level estimation. However, more detailed analysis is needed for the partition of iteration spaces. We will not discuss the detailed process here.

5 Counting heuristic

In our current implementation, a heuristic is proposed to deal with the enumeration of the interaction/union of images of parameterized polytopes.

For most applications in the image and signal processing domain, the memory space of the portion of an array touched by P1 (set of executed iterations) or P2 (set of unexecuted iterations) is usually a convex polytope that can be represented as concatenation of unoverlapped blocks. During the estimation process, we always compute the intersection/union of any two polytopes block by block and represent the result as another set of blocks. Thus, the final object for counting is also a concatenation of unoverlapped blocks. And for each block, the counting is as easy as the multiplication of the metrics in each dimension. Figure 4 shows how to compute the intersection and union of two polytopes.

Now, the computation complexity problem arises. For a set of m blocks and a set of n blocks, both the intersection and union may have at most $m \times n$ blocks. Thus, in our algorithm, the number of blocks may increase as the computation goes on. In the worst case, each block may finally degenerate into individual element. In general, suppose an array has $t1$ times write accesses (with $i_1, ..., i_{t1}$ blocks respectively) and $t2$ read accesses (with $j_1, ..., j_{t2}$ blocks respectively). Initially, the union of the $t1$ polytopes (also the union of the $t2$ polytopes) is computed. It generates two sets of blocks, with at most $i_1 \times i_2 ... \times i_{t1}$ blocks and $j_1 \times j_2 ... \times j_{t2}$ blocks each. The intersection of these two sets may have at most $i_1 \times i_2 ... \times i_{t1} \times j_1 \times j_2 ... \times j_{t2}$ blocks. Of course, the number of blocks in any set can never be larger than the size of the array.

In practice, the situation is much better due to the regular array access patterns in most image and signal processing applications:

1. The number of blocks in the initial polytopes is small. For the 5 examples we used to test the proposed algorithm, the maximum number of blocks in the initial polytopes is 3.
2. During the computation process, the increase in the number of blocks is not fast. For the examples we work on, the maximum number of blocks appearing in the final list for counting is 5.

6 Experimental results

The presented algorithm has been implemented under the SPAM framework, a compiler dealing with code optimization for retargetable compilation for embedded DSPs from Princeton University [1, 9]. And it is tested using several typical DSP applications: the auto-correlation algorithm from GSM; 2D Wavelet, motion detection; 2D DCT. The experimental results for both statement and iteration level estimation are listed in table 1. The results of statement level estimation are acquired by applying our algorithm after execution of each statement, which are the same as the minimum memory size. In the iteration level estimation, computation is performed only to the innermost loop nests. The complexity of the applications are represented by: number of arrays, number of statements (equals the number of functions in the statement level estimation, and number of loops (equals the number of functions in the iteration level estimation). The last column in table 1 gives the difference of computed memory size between the two estimation methods. From the comparison, we can see that the number of functions we need to compute (computation complexity) can be reduced by using the iteration level estimation, while the difference between the two estimation methods is always very small.

7 Related Work

Greef et al. proposed a method to do the memory size estimation for programs dealing with arrays. The method, which is based on formal mathematical model describing memory occupation of arrays, consists of two steps: intra-array and inter-array storage optimization [4]. The basic idea is to let variables share the same memory locations as much as possible. In [5], Balasa et al. described another method: arrays are partitioned into non-overlapped basic sets using operand and definition domains. A new data-flow graph is produced where the nodes are basic sets and arcs are dependences between groups of variables covered by basic sets. The nodes are weighted with the size of their corresponding basic sets and the arcs are weighted with the exact number of dependences between the basic sets corresponding to the nodes. After the partition, each basic set

applications	*arrays*	Statment Level		Iteration Level		Difference
		Memory	Function (statements)	Memory	Function (innermost loops)	
GSM Auto-correlation	4	169	7	168	3	1
Qmf_split	5	177	8	177	3	0
2D Wavelet N=16, M=10	11	832	16	832	3	0
Motion Detection M=N=32,m=n=4	5	1372	3	1372	1	0
2D DCT N=64	5	4230	9	4228	3	2

Table 1: Experimental results for some DSP applications

can be treated as a scalar. The heuristic traversal dealing with scalars can be used to determine the best possible order in which the basic sets should be produced with memory size as cost function. When a basic set is alive, it is allocated enough memory locations to store all its elements. Thus, under this scheme, they only consider memory reuse between various basic sets. The memory size can be estimated based on the size of basic sets and the number of dependences between basic sets. Due to the possible large size of basic sets, the number of memory locations computed through this method may be much higher than the absolute lower bound. To improve that, loops are unrolled before the partition process. Thus, arrays can be divided into smaller basic sets. The smaller the arrays are divided, the more exact the estimation. In the worst case, to get the absolute lower bound of memory size, the basic sets have to reduce to individual variables. In their paper, integer point counting of polytope mappings and intersection of polytope mappings is used to determine the size of basic sets and the number of dependences between basic sets (the intersection of basic sets). However, the polytopes they deal with are with constant bound. In comparison, the bounds of the polytopes in our algorithm are functions of I^*, where I^* can be any node in the iteration space. By using the proposed counting heuristic, our algorithm can get the exact memory size without unrolling the loops.

8 Conclusion and Future Work

In this paper, a new algorithm to exactly estimate the minimum memory size required by programs dealing with array computations without unrolling the loops is presented. Starting from single perfectly nested loops, our algorithm transforms the memory size estimation into an equivalent mathematical problem: integer point counting for intersection/union of mappings of parameterized polytopes. By using the proposed counting heuristic, the number of live variables after each iteration can be represented symbolically with tuples in the iteration space as free variables. Thus, the minimum memory size can be computed by finding the maximum value of the symbolic function over all iterations. Then, the method is extended to deal with imperfectly nested loops and whole programs. To decrease computation complexity, several theorems are proposed to simplify the estimation process. Experimental results on some typical DSP applications demonstrate the exactness of our algorithm. The algorithm provides the basis for comparing memory cost of different algorithms. Thus, it can be used to help making proper high-level design decisions. Further work includes considering automatic address generation for array elements to really achieve the lower bound of memory size and application of the algorithm in finding the optimal transformations given memory size as cost function.

Acknowledgments

This research was funded by DARPA and the New Jersey Center for Multimedia Research.

References

[1] A.Sudarsanam. Code optimization libraries for retargetable compilation for embedded digital signal processors. Phd thesis, Princeton University, May 1998.

[2] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. *10th ACM Int. Conf. on Supercomputing*, May 1996.

[3] P. Clauss. Handling memory cache policy with integer points countings. *Euro-Par'97*, pages 285–293, 1997.

[4] H. M. E.De Greef, F.Catthoor. Array placement for storage size reduction in embedded multimedia systems. *11th International Conference on Application-specific Systems, Architectures and processors*, July 1997.

[5] H. D. M. F. Balasa, F. Catthoor. Background memory area estimation for multi-dimensional signal processing systems. *IEEE Trans. on Comp-aided Design*, CAD-14, 1995.

[6] A. F.J.Kurdahi. Real: a program for register allocation. *Proc. 24th DAC*, pages 210–215, June 1987.

[7] C. Lengauer. Loop parallelization in the polytope model. in e.best. *CONCUR'93, Lecture Notes in Computer Science 715*, pages 398–416, 1993.

[8] W. Pugh. Counting solutions to presburger formulas: How and why. *Proc. of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[9] A. Sudarsanam and S. Malik. Simultaneous reference allocation in code generation for dual data memory bank asips. *To be published in ACM Transactions on Design Automation for Electronic Systems*, 1999.