Estimating and Reducing the Memory Requirements of Signal Processing Codes for Embedded Systems

J. Ramanujam, *Member, IEEE,* Jinpyo Hong, Mahmut Kandemir, and A. Narayan

***Abstract*— Most embedded systems have limited amount of memory. In contrast, the memory requirements of the DSP and video processing codes (in nested loops, in particular) running on embedded systems is significant. This paper addresses the problem of estimating and reducing the amount of memory needed for transfers of data in embedded systems. First, the problem of estimating the region associated with a statement or the set of elements referenced by a statement during the execution of nested loops is analyzed. For a fixed execution ordering, a quantitative analysis of the number of elements referenced is presented; exact expressions for uniformly generated references and a close upper and lower bound for non-uniformly generated references are derived. Second, in addition to presenting an algorithm that computes the total memory required, this paper also discusses the effect of transformations (that change the execution ordering) on the lifetimes of array variables, i.e., the time between the first and last accesses to a given array location. The term *maximum window size* is introduced and quantitative expressions are derived to compute the maximum window size. A detailed analysis of the effect of unimodular transformations on data locality including the calculation of the maximum window size is presented.**

***Index Terms*— Memory estimation, signal processing codes, embedded processors, data reuse, execution ordering, compiler transformations.**

I. Introduction

An important characteristic of embedded systems that execute signal and image processing codes is that the hardware can be customized according to the needs of a single or a small group of applications. An example of such customization is parameterized memory/cache modules whose topological parameters (e.g., total capacity, block size, associativity) can be set depending on the data access pattern of the application at hand. In many cases, it is most beneficial to use the smallest amount of data memory that satisfies the target performance level [18]. Employing a data memory space that is larger than necessary has several negative consequences. First, the per access energy consumption of a memory module increases with its size [2]. Second, larger memory modules tend to incur larger delays, thereby increasing the data access latency. Third, large memories by definition occupy more chip space. Consequently, significant savings in energy, area and delay may be possible through a more careful selection of memory size.

Unfortunately, selecting the minimum memory size (without impacting performance) is not always easy. This is because data declarations (which specify the total size of each array, for example) in many codes are decided based on high-level representation of the algorithm being coded not based on actual memory requirements. The important point here is that not every data item (declared data location) is needed throughout the execution of the program. That is, at any given point during the execution, typically, only a portion of the declared storage space (e.g., array size) is actually needed. This is particularly true for embedded systems that process data arrays in signal processing codes. Therefore, the total data memory size can be reduced by determining the *maximum* number of data items that are *live* at any point during the course of execution. Two complementary steps to achieve this objective is (i) estimating the memory consumption of a given code, and (ii) reducing the memory consumption through access pattern transformations.

The problem of estimating the minimum amount of memory for a fixed execution ordering of nested loops was recently addressed by Zhao and Malik [18]. Their work characterizes the memory estimation problem as one of counting the number of live variables in each iteration of the innermost loop of loop nest—which is prohibitively expensive for large multi-dimensional arrays—and deriving the maximum of these. In this paper, we present a technique that quickly and accurately estimates the number of distinct array accesses and the minimum amount of memory in nested loops, which does not involve calculations at each iteration of a nested loop. Moreover, we present a technique that reduces the amount of memory required through the use of loop-level transformations. See the section on related work for a comparison of our approach to those in other works [4], [9], [11], [12], [18].

Nested loops are of particular importance as many embedded codes from image and video processing domains manipulate large arrays using several nested loops. In most cases, the number of distinct accesses is much smaller than the size of the array(s) in question and the size of the loop iteration space. This is due to the repeated accesses to the same memory location in the course of execution of the loop nest. The proposed technique identifies this reuse of memory locations, and takes advantage of it in estimating the memory consumption as well as in reducing it.

The main abstraction that our technique manipulates is that of *data dependence* and *data re-use* [17]. Since many compilers that target array-dominated codes maintain some sort of data dependence information, implementing our estimation and optimization strategy involves only a small additional

Manuscript received December 28, 2001; revised May 12, 2003. This work was supported in part by NSF Young Investigator Award 9457768, and by NSF grants 0073800, 0093082 and 0103693.

J. Ramanujam and A. Narayan are with Louisiana State University. Email: {jxr,hpc}@ece.lsu.edu. Jinpyo Hong is with University of Memphis. Email: jphong1@memphis.edu. Mahmut Kandemir is with Pennsylvania State University. Email: kandemir@cse.psu.edu.

overhead. Our experimental results obtained using a set of seven codes show that the proposed techniques are very accurate, and are capable of reducing the memory consumption significantly through high-level optimizations.

The rest of this paper is organized as follows. Section 2 presents a brief background. Section 3 presents our techniques for estimating the number of distinct references to arrays accessed in nested loops (as a function of loop bounds and reuse distances) in which the execution ordering is fixed to be the sequential execution ordering. In Section 4, we present a loop transformation technique that finds the best execution ordering that minimizes the maximum amount of memory required. A general method of deriving the transformation is presented. Section 5 presents experimental results on seven benchmarks from signal and image processing domains. Related work is discussed in Section 6, and Section 7 concludes with a summary.

II. Background

DSP programs mainly consist of perfectly nested loops (loop nests in which every statement is inside the innermost loop) that access single and multi-dimensional arrays [17], [18]. Therefore, we will limit our discussion to these cases. In our framework, the execution of each iteration of an n-level nested loop is represented using an *iteration vector* $\vec{I} = (I_1, \cdots, I_n)$, where I_j corresponds to the j^{th} loop from the outermost position. We assume that the array subscript expressions and loop bounds are *affine functions* of enclosing loop indices and loop-independent parameters or variables [17]. Each reference to a d-dimensional array U is represented by an *access* (or *data reference*) *matrix* A_D and an *offset vector* $\vec{b}$ such that $A_D\vec{I} + \vec{b}$ is the element accessed by a specific iteration $\bar{I}$ [17]. The access matrix is a $d \times n$ matrix. An array element which is referenced (read or written) more than once in a loop nest constitutes a *reuse.* The degree of reuse (i.e., reuse count) depends on the number of references to the array in the loop (r), the relative values of the loop nest levels and the dimensionality (d) of the array, and also the loop limits. If iterations $\vec{i}$ and $\vec{j}$ access the same location, we say that the *reuse vector* is $\vec{j} - \vec{i}$. The level of a reuse vector is the index of the first non-zero in it. Consider the following loop nest.

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        for $k = 1$ to $N$ do
            $\cdots U[i, k-3] \cdots$
```

The iteration vector is $(i, j, k)^T$ (note that we write a column vector as transpose of the corresponding row vector), the data access matrix for array U is $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ and the offset vector is $\begin{pmatrix} 0 \\ -3 \end{pmatrix}$.

For an array whose reference matrix is square (that is, if the dimension d of the array and the deepest loop nesting level n are the same), the number of times an element is referenced is at most r where r is the number of references to the array in a loop. Therefore the *reuse* for a data element is at most $r-1$. For an array whose dimension is one less than the deepest loop nesting level, the reuse of an element is along the direction of the null space of the access matrix and the amount of reuse due to an element depends on the loop bounds. In the example given above, the reuse is in the direction of the middle loop index (j). Arrays whose number of dimensions is one less than the depth of the nested loop enclosing them are common in DSP applications [2]. For d-dimensional arrays accessed in n-nested loops where $d < n$ in general, the extent of reuse depends on the loop bounds as well the basis vectors of the null space of the access matrix.

A. Data Dependences and Loop Transformations

Dependence and reuse analysis are critical to the success of optimizing compilers [16], [17]. We deal with sets of perfectly nested loops, whose upper and lower bounds are all linear, enclosing a loop body with affine array references. That is, each subscript of an array variable index expression must be an affine expression over the scalar integer variables of the program. We assume familiarity with definitions of the types of dependences [17].

Dependences arise between two iterations $\vec{i}$ and $\vec{j}$ when they both access the same memory location and one of them writes to the location [17]. Let $\vec{i}$ execute before $\vec{j}$ in sequential execution; the vector $\vec{d} = \vec{j} - \vec{i}$ is referred to the *dependence vector* [17]. This forces sequentiality in execution. The level of a dependence vector is the index of the first non-zero element in it [17]. Let $\vec{i} = (i_1, \ldots, i_n)$ and $\vec{j} = (j_1, \ldots, j_n)$ be two iterations of a nested loop such that $\vec{i} \prec \vec{j}$ (read $\vec{i}$ precedes or executes before $\vec{j}$ in sequential execution) and there is a dependence of constant distance $\vec{d}$ between them. Applying a linear transformation T to the iteration space (nest vector) also changes the dependence matrix since $T(\vec{j}) - T(\vec{i}) = T(\vec{j} - \vec{i}) = T\ \vec{d}$. All dependence vectors are *positive vectors,* i.e., the first non-zero component should be positive. We do not include *loop-independent* dependences which are zero vectors. A transformation is legal if the resulting dependence vectors are still *positive* vectors [17]. Thus, the linear algebraic view leads to a simpler notion of the legality of a transformation. For an n-nested sequential loop, the $n \times n$ identity matrix (I_n) denotes sequential execution order. Any unimodular transformation can be realized through reversal, interchange and skewing [16], [17].

B. Distinct References

The number of distinct references (A_d) can be found using dependences in the loop as shown in Figure 1. The n-dimensional cube (in the case of 2-nested loop, this is a square) formed by the dependence vector $(3, -2)$ as shown in Figure 1 represents the reused area (the shaded area) in the iteration space. The dependence (reuse also)vector implies that the iterations (i, j) and $(i+3, j-2)$ access the same memory location $A[i, j]$. The array element $A[i, j]$ is live between iterations (i, j) and $(i+3, j-2)$ in sequential execution. Let us suppose that the array element $A[i, j]$ is mapped to some memory location x, and that the second reference is a write in Example 1(a) shown below. This means that after the value in $A[i, j]$ is used in iteration $(i+3, j-2)$, the memory

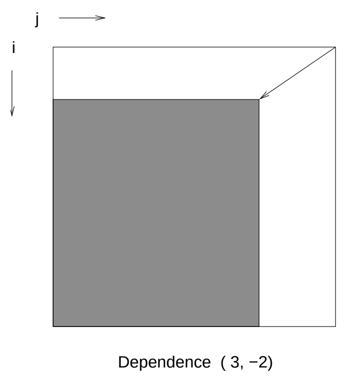

Fig. 1. Iteration space for a 2-nested loop.

location x can hold other array elements. In fact, x can hold all the values of the array elements written in iterations $(i, j), (i+3, j-2), (i+6, j-4), \cdots, (i+3t, j-2t), \cdots$ where t is any integer. Every iteration in the non-shaded region of the iteration space accesses a distinct array element; the memory requirement is the area of the non-shaded region. Note that every iteration in the shaded region of the iteration space reuses an array element. Consider the following examples:

Example 1(a):

```
for $i = 1$ to $10$ do
    for $j = 1$ to $10$ do
        $\cdots A[i-3, j+2] \cdots$
        $\cdots A[i, j] \cdots$
```

Example 1(b):

```
for $i = 1$ to $10$ do
    for $j = 1$ to $10$ do
        $\cdots A[2 * i + 3 * j] \cdots$
```

In both Example 1(a) and 1(b), the dependence vector is $(3, -2)$. Note that in the first example, the dimensionality of the array is the same as that of the loop nest level, the number of references (r) is 2 and the reuse count is at most 1. (The number of times an element of the array is referenced is at most 2).

In the second example, the dimensionality is less than the loop nest level, the number of references (r) is 1, and the maximum reuse count for an element is $\lceil 10/3 \rceil$ = 4. The total reuse (i.e., the area of the shaded region) is the same in both the examples which is $(10-3) \times (10-2) = 56$. Let the dependence vector be (d_1, d_2). In general, the signs of d_1 and d_2 do not affect the amount of reuse. In a nested loop of size $N_1 \times N_2$, the amount of reuse is given by $(N_1 - |d_1|) \times (N_2 - |d_2|)$. The total number of distinct locations accessed (the minimum memory size), therefore, is $N_1 \times N_2 - (N_1 - |d_1|) \times (N_2 - |d_2|) = N_1 \times |d_2| + N_2 \times |d_1| - |d_1| \times |d_2|$. Note that in contrast to other works [4], [9], [11], [12], [18] that require the loop bounds to be known constants, we derive closed-form expressions for the minimum memory size as a function of the loop bounds and dependence vectors. We consider the cases where the dimension of the array accessed within the loop is the same as the nest level and where the dimensionality is less than the loop nesting level. It should be emphasized that these cases are commonly found in DSP codes [2].

C. Uniformly Generated References and Maximum Window Size

We assume that all the references to an array are *uniformly generated* [5], [7]. Uniformly generated references are those, for which the access matrices are the same but the offset vectors are different, i.e., the subscript functions of the different references differ only in the constants. To illustrate this concept, consider the following loop.

```
for $i_1 = l_1$ to $u_1$ do
    $\cdots$
        for $i_n = l_n$ to $u_n$ do
            $\mathcal{R}_1[A_1(i_1, \ldots, i_n) + \vec{b_1}]$
            $\cdots$
            $\mathcal{R}_k[A_k(i_1, \ldots, i_n) + \vec{b_k}]$
```

where $\mathcal{R}_1, \ldots, \mathcal{R}_k$ are references to arrays not necessarily distinct. The A_i are matrices of size $n \times d_i$ where d_i is the dimension of the array referenced in $\mathcal{R}_i$; $\vec{b_i}$ is a vector of size d_i. We assume that all array reference to an array X are uniformly generated, i.e., A_i for all references to X are equal. An example of a loop with a uniformly generated references is shown below:

```
for $i = 1$ to $N_1$ do
    for $j = 1$ to $N_2$ do
        $X[\boxed{2i+3j}+2] = Y[\boxed{i+j}]$
        $Y[\boxed{i+j}+1] = X[\boxed{2i+3j}+3]$
```

Here, the two references to X are of the form $\boxed{2i + 3j + \text{constant}}$ and both references to array Y are of the form $\boxed{i + j + \text{constant}}$. We use the notion of a reference window of an array in loop nest (which is different form the notion of the reference window of a dependence as used by [5], [7]) that allows us to deal with each distinct array as a whole and not on a per-reference-pair-to-the-array basis.

The amount of memory required is a function of the number of *live variables*, i.e., variables that will be accessed again in the future. We now introduce a notion that is useful in this context. The *reference window* $W_X(\vec{i})$ (where $\vec{i} = (i_1, \ldots, i_n)$ is an iteration of the n-nested loop) is the set of all elements of array X that are referenced by any of the statements in all iterations $\vec{j_1} \preceq \vec{i}$ (read $\vec{j_1}$ precedes in sequential execution or is the same as $\vec{i}$) that are also referenced in some (later) iteration $\vec{j_2}$ such that $\vec{j_2} \succ \vec{i}$ (read $\vec{j_2}$ follows $\vec{i}$). This allows us to precisely the define those iterations which need a specific value in local memory. The size of the window $W_X(\vec{i})$ is the number of elements in that window. The *maximum window size* (MWS) is given by

$$\max_{\vec{i}} \left| W_X(\vec{i}) \right|$$

and is defined over the entire iteration space. In the case of multiple arrays $X_1, \ldots, X_K$, the maximum reference window size is:

$$\max_{\vec{i}} \sum_{k=1}^{K} \left| W_{X_k}\left(\vec{i}\right) \right| .$$

Note that the reference window is a *dynamic* entity, whose shape and size change with execution. For nested loops with uniformly generated references, the maximum window size (MWS) is a function of the loop limits. The smaller the value of MWS, the higher the amount of data locality in the loop nest for the array. For simplicity of exposition, we assume that there are multiple uniformly generated references to a single array in a loop nest. The results derived here easily generalize to multiple arrays and higher levels of nesting.

III. Estimating the Number of Distinct Accesses in Nested Loops for Fixed (Sequential) Execution Ordering

The amount of memory needed is a function of the access pattern. In this section, we develop estimates of the memory requirement in terms of the total number of distinct accesses in a nested loop, assuming no transformations are employed and no dynamic memory management is used; dynamic memory management would allow one to use the same memory location for accesses to two variables whose lifetimes (the time spanned from the first access to a location to the last access) are disjoint. These are discussed in Section 4.

A. Loops with Array Dimension (d) = Nesting (n)

With just one reference to each array in such a nest, the number of distinct accesses equals the total number of iterations. Therefore, we focus only on the case where there are multiple references to the same array. Stencil codes including some relaxation codes such as SOR exhibit such an access pattern.

In general for r references in a loop where the array dimension is the same as the loop nesting level there are a total of $\frac{r(r-1)}{2}$ dependences. Note that there is at least one node in the dependence graph which is a sink to the dependence vectors from each of the remaining $r-1$ nodes. In other words, there exists a statement with $r-1$ direction vectors directed from each of the remaining statements. The $r-1$ dependences due to all the other references to this reference gives the amount of reuse. Consider a two-level nested loop in which there are r uniformly generated references. Let the dependences on one reference due to all other references be

$$\begin{pmatrix} d_{11} & d_{21} & \cdots & d_{r-1,1} \\ d_{12} & d_{22} & \cdots & d_{r-1,2} \end{pmatrix}.$$

The amount of reuse for that array is:

$$\text{reuse} = \sum_{i=1}^{r-1} (N_1 - |d_{i1}|)(N_2 - |d_{i2}|)$$

and the number of distinct elements is given by

$$A_d = N_1 \times N_2 \times r - \text{reuse}.$$

Consider the following loop (in Example 2) where there are two uniformly generated references to the array A and the access matrix is non-singular.

Example 2:

```
for $i = 1$ to $N_1$
    for $j = 1$ to $N_2$
$S_1$:      $\cdots A[i,j] \cdots$
$S_2$:      $\cdots A[i-1, j+2] \cdots$
```

Here there is a dependence $(1,-2)$ from statement S_1 to statement S_2. This dependence is used to calculate the amount of reuse for each element. The amount of reuse is $(N_1 - 1)(N_2 - 2)$, and the number of distinct accesses to the array A in the above loop is $A_d = N_1 \times N_2 \times 2 -$ reuse.

Example 3 (shown below) illustrates the case of several uniformly generated references.

Example 3:

```
for $i = 1$ to $10$
    for $j = 1$ to $10$
$S_1$:      $\cdots A[i,j] \cdots$
$S_2$:      $\cdots A[i-1,j] \cdots$
$S_3$:      $\cdots A[i,j-1] \cdots$
$S_4$:      $\cdots A[i-1,j-1] \cdots$
```

The dependence vectors from statement S_1 to the other statements are $(1,0),(0,1),(1,1)$. The amount of reuse is calculated as reuse $= (10-1)(10-0) + (10-0)(10-1) + (10-1)(10-1) = 90 + 90 + 81 = 261$, and the the number of distinct accesses is: $A_d = 10 \times 10 \times 4 - 261 = 139$. Thus, we see that, for cases where the loop nesting level is the same as the dimension of the array accesses in the loop, there is only one dependence vector between a pair of statements and the maximum reuse for a particular element is at most $r-1$. In other words, there are a maximum of r references to any given array element.

B. Loops with Array Dimension $d = n - 1$

a) Single Reference: Consider the case of a single reference where the dimension of the array is at least one less than the loop nest. If $d = n-1$, then there is reuse along the direction of the null space vector of the access matrix.

Example 4:

```
for $i = 1$ to $20$ do
    for $j = 1$ to $10$ do
        $\cdots A[2i+5j+1] \cdots$
```

Here the reuse vector is $(5,-2)$ which is the same as the dependence vector for the loop. We now look at the n dimensional cube formed by the dependence vector (in this case, a square) on the iteration space which represents the reused elements of the array. Note that all elements within the square formed by the vector is a sink to a direction vector which is a reused element by definition. Therefore, for the above example where there is a single statement, we can obtain the figure for the number of data elements reused in the array as:

$$\text{reuse} = (N_1 - d_{11})(N_2 - |d_{21}|) \;=\; (20-5)(10-2) = 120,$$

and the number of distinct accesses to the array is

$$A_d = N_1 \times N_2 - \text{reuse} = 20 \times 10 - 120 = 80.$$

Now consider the case of a 2-dimensional array accessed in a three-level nested loop.

Example 5:

```
for $i = 1$ to $10$ do
    for $j = 1$ to $20$ do
        for $k = 1$ to $30$ do
            $\cdots A[3i+k, j+k] \cdots$
```

Here the reuse vector is $(1, 3, -3)$; the reuse is calculated as:

$$\text{reuse} = (10-1)(20-3)(30-3) = 4131,$$

and the number of distinct accesses is

$$A_d = 10 \times 20 \times 30 - 4131 = 1869.$$

Extensions to handle the case of multiple uniformly generated references and general classes of references are beyond the scope of this paper.

IV. Minimizing the Maximum Window Size Using Transformations

The last section presented estimates of the memory requirement in terms of the total number of distinct accesses in a nested loop, assuming no transformations are employed (fixed execution ordering) and no dynamic memory management is used. In this section, we show how to derive program transformations that can be used to reduce the size of the maximum amount of memory required. Of course, this assumes a dynamic memory management scheme that would allow us to use the same memory location for accesses to two variables whose lifetimes are disjoint.

Consider the following example which is a minor variant of the example from [5]:

Example 6:

```
for $i = 1$ to $20$ do
    for $j = 1$ to $30$ do
        $\cdots X[2i - 3j] \cdots$
```

Eisenbeis et al. [5] mention that the cost of the window (the same as MWS) for this loop is 89. They use only two transformations: loop interchange and reversal. On applying interchange, the MWS reduces to 41. On reversal applied to the original loop, the cost becomes 86 while reversing the interchanged loop reduces the cost to 36. Using the technique presented here, the cost or MWS for this loop can be reduced to 1, i.e., all iterations accessing any element of the array X can be made consecutive iterations of an inner loop. The only dependence in this example is the vector $(3, 2)$. We use the following legal transformation, $T = \begin{pmatrix} 2 & -3 \\ 1 & -1 \end{pmatrix}$. Note that the first row of the transformation matrix $(2\ -3)$ is the same as the coefficients of the access function $2i - 3j$. Li and Pingali's technique [14] constructs the transformation matrix using the rows of the access matrix. Even though the technique in [14] can be used to derive this transformation, there are situations where the techniques presented here *improves* locality while that in [14] does not improve locality. Consider the loop shown in the next example.

Example 7:

```
for $i = 1$ to $25$ do
    for $j = 1$ to $10$ do
        $X[2i + 5j + 1] = X[2i + 5j + 5]$
```

The distance vectors for this loop are: $(3, -2), (2, 0), (5, -2)$; $(3, -2)$ is the flow dependence, $(2, 0)$ is an anti-dependence and $(5, -2)$ is the output dependence vector. These are the only direct dependences. Li and Pingali use transformation matrices whose first row is either $(2, 5)$ or $(-2, -5)$. Any transformation that uses $(2, 5)$ as its first row is illegal because of the distance vector $(3, -2)$; the first component of $(3, -2)$ after the transformation is $((2, 5) \cdot (3, -2)^T = -4$ is $< 0)$. Similarly any transformation that uses $(-2, -5)$ as its first row is illegal due to the distance vector $(2, 0)$ since $((-2, -5) \cdot (2, 0)^T = -4$ is $< 0)$. The maximum window size is 50. Li and Pingali's technique will not find any partial transformation that can be completed to a legal transformation. Where as, by applying techniques presented in the following sections, we can apply the legal transformation, $T = \begin{pmatrix} 2 & 3 \\ 1 & 1 \end{pmatrix}$. Applying T reduces the maximum window size to 21. A combination of reversal and interchange does not change the maximum window size from 50.

A. Effect of Transformations on Locality

Consider a nested loop with r uniformly generated references to an array X of the form: $\lambda_1 i + \lambda_2 j + c_k$ $(k = 1, \ldots, r)$ as shown below:

Example 8:

```
for $i = 1$ to $N_1$ do
    for $j = 1$ to $N_2$ do
        $\cdots X[\lambda_1 i + \lambda_2 j + c_1] \cdots$
        $\cdots$
        $\cdots X[\lambda_1 i + \lambda_2 j + c_r] \cdots$
```

We need to compute the effect of a legal unimodular transformation, $T = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ on the maximum window size. In addition to legality, we require that the loop nest be *tile-able* [10], [16]; this permits us to use block transfers, which are very useful to minimize the number of off-chip accesses. The optimum transformation thus satisfies two conditions:

1) legality condition for tiling
2) minimizes the maximum window size

The detailed derivation is beyond the scope of this paper. The maximum window size (MWS) is a function of the maximum inner loop span or *maxspan*, which is the maximum trip count of the inner loop (difference between the upper and lower limits of the inner loop) over all outer loop iterations [5].

$$\text{MWS} = \text{maxspan} \times \Delta \times (\lambda_2 a - \lambda_1 b) \tag{1}$$

where Δ is the determinant of the transformation matrix. Let $\theta = \lambda_2 a - \lambda_1 b$. The simplified expression derived for the maximum window size is:

$$MWS = \begin{cases} \left(\left|\frac{N_1 - 1}{b}\right| + 1\right) |\theta| & \text{if } a - b \geq aN_1 - bN_2 \\ \left(\left|\frac{N_2 - 1}{a}\right| + 1\right) |\theta| & \text{if } a - b \leq aN_1 - bN_2 \end{cases} \tag{2}$$

Thus to minimize the maximum window size, the value of MWS from equation (2) should be minimized among all unimodular transformations T that are valid for tiling. In many cases, MWS is minimized when $|\lambda_2 a - \lambda_1 b|$ is minimized.

B. Legal Transformation

Let $\vec{d_i} = (d_{i,1}, d_{i,2})$ $(i = 1, \ldots, m)$ be a set of dependence distance vectors. With uniformly generated references, all the dependences in a nested loop are distance vectors. Given any two uniformly generated references $\lambda_1 i + \lambda_2 j + c_1$ and $\lambda_1 i + \lambda_2 j + c_2$, to test for a dependence from iteration (i_1, i_2) to iteration (j_1, j_2), we check for integer solutions within the loop range to the equation:

$$\lambda_1 i_1 + \lambda_2 i_2 + c_1 = \lambda_1 j_1 + \lambda_2 j_2 + c_2$$
$$\text{i.e.,} \qquad \lambda_1(j_1 - i_1) + \lambda_2(j_2 - i_2) = c_1 - c_2.$$

We can write $x_1 = j_1 - i_1$ and $x_2 = j_2 - i_2$ where (x_1, x_2) is a distance vector. Since $\lambda_1, \lambda_2, c_1, c_2$ are constants, every solution gives a distance vector. The smallest lexicographically positive solution is the dependence vector of interest. In order for the transformation T to render the loop nest tile-able, the following conditions must hold:

$$ad_{i,1} + bd_{i,2} \geq 0 \qquad i = 1, \cdots, m$$
$$cd_{i,1} + dd_{i,2} \geq 0 \qquad i = 1, \cdots, m$$

We illustrate the use of technique through Example 2. Consider the loop nest:

```
for $i = 1$ to 25 do
    for $j = 1$ to 10 do
        $X[2i + 5j + 1] = X[2i + 5j + 5]$
```

The distance vectors for this loop are: $(3, -2), (2, 0), (5, -2)$. The problem here is to find a unimodular transformation: $T = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ such that the loop is tile-able (which allows bringing chunks of data which can fully operated upon before discarding), *i.e.*, represented by the following constraints:

- $3a - 2b \geq 0$, $2a \geq 0$, $5a - 2b \geq 0$, $3c - 2d \geq 0$, $2c \geq 0$, $5c - 2d \geq 0$.

and the maximum window size (MWS)

$$MWS = \begin{cases} \left(\left|\frac{24}{b}\right| + 1\right)|5a - 2b| & \text{if } a - b \geq 25a - 10b \\ \left(\left|\frac{9}{a}\right| + 1\right)|5a - 2b| & \text{if } a - b \leq 25a - 10b \end{cases}$$

is minimized. Given the set of inequalities that should be satisfied,

$$3a - 2b \geq 0 \Longrightarrow b \leq \frac{3a}{2} \Longrightarrow 9b \leq \frac{27a}{2}.$$

Since, $9b \leq \frac{27a}{2}$, the second condition applies, *i.e.*, $9b \leq 24a$. So,

$$MWS = \left(\frac{9}{a} + 1\right)(5a - 2b) = 45 + (5a - 2b) - \frac{18b}{a}$$

needs to be minimized subject to inequalities (2.5–2.10). We use either a branch and bound technique (or general nonlinear programming techniques) to minimize this function; the number of variables is linear in the number of nested loops which is usually very small in practice (≤ 4) resulting in small solution times. Alternately, if we minimize $5a - 2b$ subject to constraints (2.5–2.10), we get very good solutions in practice. In the example loop nest, $a = 2, b = 3$ is an optimal solution, giving an minimum MWS estimate of 22 which is very close to the actual minimum MWS which is 21. In general, the system of inequalities arising legal tiling requirement are combined with either $a - b \leq aN_1 - bN_2$ or with $a - b \geq aN_1 - bN_2$ to form two groups of inequalities; if both the groups have valid solutions, we find the best of these. If only one group has valid solutions, the problem is a lot easier. For the solution $a = 2, b = 3$, the set of values for c and d which give rise to unimodular T while satisfying tiling legality condition is $c = 1, d = 1$.

The window size in 3-nested loops cannot be just derived using the coefficients of the access functions and is a function of the null space vector and the loop limits. It is estimated using the largest lexicographic dependence vector.

V. Experimental Results on Signal, Image and Video Processing Codes

In order to evaluate the proposed estimation and optimization technique, we tested it using seven codes from DSP and video processing domains: `2_point` and `3_point` are two-point and three-point stencil codes, respectively; `sor` is a successive-over-relaxation code; `matmult` is a matrix-multiply kernel; two different motion estimation codes, `3step_log` and `full_search`; and finally, `rasta_flt` is a filtering routine from MediaBench [13]

Figure I presents our results in columns 2 through 5. The column `default` gives the normal memory size which is the total number of array elements declared. $\texttt{MWS}_{unopt}$ and $\texttt{MWS}_{opt}$, on the other hand, give the maximum window sizes (MWS) before and after optimizing the code, respectively. In columns 3 and 4, following each number, within parentheses, we also give the *percentage reduction* with respect to the corresponding value in the second column. Column 5 denotes the ratio of $\texttt{MWS}_{opt}$ to $\texttt{MWS}_{unopt}$ as a percentage. We see from these results that estimating the memory consumption (requirements) of the original (unoptimized) codes indicates a 81.9% saving, and that for the optimized codes brings about an average saving of 92.3%. Note that these savings directly correspond to reduction in the required data memory sizes. We also need to mention that except for `rasta_flt`, our estimations were exact. In the `rasta_flt` code, our estimation is around 13% higher than the actual memory requirement for both the original and the optimized code. It is useful to note that except for `matmult`, we see a significant reduction in memory through the use of transformations.

VI. Related work

The estimation of the number of references to an array in order to predict cache effectiveness in hierarchical memory machines have been discussed by Ferrante et al. [6] and Gallivan et al. [7]. The image of the iteration space onto the array space to optimize global transfers have been discussed in [7]. A framework for estimating bounds for the number of elements accessed only was given. Ferrante et al. gave exact values for uniformly generated references but did not consider multiple references. Also, for non-uniformly generated references, arbitrary correction factors were given for arriving at lower and upper bounds for the number of distinct references. We present a technique in this paper which gives accurate results for most practical cases and very close bounds where ever necessary. Clauss [3] and Pugh [15] have presented more

TABLE I

DEFAULT AND ESTIMATED MEMORY REQUIREMENTS FOR SIGNAL, IMAGE AND VIDEO PROCESSING CODES.

code	default	MWS$_{unopt}$	MWS$_{opt}$	$\frac{\text{MWS}_{opt}}{\text{MWS}_{unopt}}$
2_point	4,096	65 (98.4%)	3 (99.9%)	4.6%
3_point	1,024	68 (93.3%)	35 (96.5%)	51.5%
sor	1,024	65 (93.6%)	35 (96.5%)	53.8%
matmult	768	273 (64.4%)	273 (64.4%)	100.0%
3step_log	2,064	511 (75.2%)	122 (94.0%)	23.9%
full_search	2,064	252 (87.8%)	60 (97.1%)	23.8%
rasta_flt	5,152	2,040 (60.4%)	127 (97.5%)	6.2%
Average Reduction:		81.9%	92.3%	
Average Ratio:				37.7

expensive but exact techniques to count the number of distinct accesses.

Some early approaches in high-level synthesis that dealt with minimum register allocation for scalars [8] can be extended to arrays by treating each array element as a separate scalar; such an approach is highly expensive. Researchers from IMEC [1], [4], [11], [12], Grun et al. [9], and Zhao and Malik [18] present techniques that estimate the minimum amount of memory required. Of these [11], [12] allow the user to specify an arbitrary execution ordering, Balasa et al. [1] ignores execution ordering, while the rest of the works assume a fixed sequential execution ordering.

Balasa et al. [1] do not take into account the effects of execution ordering in deriving memory estimates; their work assumes that all loop bounds are given constants and results in significant over-estimation. De Greef et al. [4] consider *in-place mapping* which exploits non-overlapping life times of arrays and array elements to reduce the overall memory storage, for a given execution ordering. They also require that all loop bounds are constants. We do not consider the effects of array layouts and interleaving in this paper. Our first goal is to reduce the the memory needs of individual arrays by reducing their maximum window sizes through loop transformations. We are currently exploring the use of *in-place mapping.*

Grun et al. [9] assume fixed execution ordering and constant loop bounds and derive estimates of upper and lower bounds on the memory requirement of codes by using the first and last iterations of each loop in a nest as starting points and further refining by using additional iterations as needed to get better estimates. In the worst case, complete unrolling of nested loops may be required. In contrast, we present closed form expressions for memory requirement and derive parameterized choice of loop transformations in order to minimize the memory requirement.

Zhao and Malik [18] addressed the problem of estimating the minimum amount of memory for a fixed execution ordering of nested loops. They view memory estimation as counting the number of live variables in each iteration of the innermost loop of loop nest—which is prohibitively expensive for large multi-dimensional arrays—and deriving the maximum of these. Our work differs from Zhao and Malik's [18] in that we present analytical expressions for memory requirement that do not involve calculations for each iteration of any loop. Also, our work addresses loop transformations to reduce memory.

Kjeldsberg et al. [11], [12] allow users to specify partial execution orderings and assume that the loop bounds are known constants. They are able to apply their work to imperfectly nested loops as well by deriving the notion of a common iteration space of a set of loops. We note that there is no unique way to construct such a common iteration space, and approaches to this problem have so far been ad hoc [17]. Different choices of common iteration space lead to different memory requirements. Kjeldsberg et al. [11], [12] also assume that the whole code is in static single assignment form which is not a requirement in our case.

None of the above works discuss how to derive transformations that reduce the amount of minimum memory required. In addition, in contrast to these works, we present closed form expressions for memory reuse and derive the memory requirement as a function of loop transformations.

Our work on loop transformations for improving data locality bears most similarity to work by Gannon et al. [7] and Eisenbeis et al. [5]. They define the notion of a reference window for each dependence arising from uniformly generated references. Unlike our work, they do not use compound transformations – only interchange and reversal are considered. In addition, the use of a reference window and the resultant need to approximate the combination of these windows results in a loss of precision. Ferrante et al. [6] present a formula that estimates the number of distinct references to array elements; their technique does not use dependence information. Wolf and Lam [16] develop an algorithm that estimates temporal and spatial reuse of data using *localized vector space.* Their algorithm combines reuses from multiple references. Their method does not use loop bounds and the estimates used are less precise than the ones presented here. Their method performs an exhaustive search of loop permutations that maximizes locality. Li and Pingali [14] discuss the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are selected subscript functions for various array accesses (excluding constant offsets). While their technique exploits reuse arising from input and output dependences, it does not work well with flow or anti-dependences.

VII. SUMMARY

Minimizing the amount of memory required is very important for embedded systems. The problem of estimating the minimum amount of memory was recently addressed by Zhao and Malik [18]. In this paper, we presented techniques that (i) quickly and accurately estimates the number of distinct array accesses and the minimum amount of memory in nested loops, and (ii) reduces this number through loop-level transformations. The main abstraction that our technique manipulates is that of data dependence and re-use [17]. Since many compilers that target array-dominated codes maintain some sort of data dependence information, implementing our estimation and optimization strategy involves only a small additional overhead. Our experimental results obtained using a set of seven codes show that the proposed techniques are very accurate, and are capable of reducing the memory consumption significantly through high-level optimizations. Work is in progress to extend our techniques to include the effects of memory layouts of arrays and inter-array in-place mapping, to extend the scope of transformations used, and to increase the applicability of our solution to whole programs.

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, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology.* Kluwer Academic Publishers, June 1998.

[3] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. ACM Int. Conf. Supercomp.,* May 1996.

[4] E. De Greef, F. Catthoor and H. De Man. Array placement for storage size reduction in embedded multimedia systems. *Proc. 11th Int. Conf. Application-Specific Systems, Architectures and Processors,* pp. 66–75, July 1997.

[5] C. Eisenbeis, W. Jalby, D. Windheiser and F. Bodin. A strategy for array management in local memory. *Advances in Languages and Compilers for Parallel Computing,* pp. 130–151, 1991.

[6] J. Ferrante, V. Sarkar and W. Thrash. On Estimating and Enhancing Cache Effectiveness. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing,* August 1991.

[7] K. Gallivan, W. Jalby and D. Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. *Proc. ACM Int. Conf. Supercomp.,* pp. 238–253, 1988.

[8] C. Gebotys, M. Elmasry. Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis. In *Proc. Design Automation Conference,* pp. 2–7, June 1991.

[9] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Proc. of the 6th IEEE/ACM Int. Workshop on Hardware/Software Co-Design (CODES/CASHE '98),* pp. 145-149, March 1998.

[10] F. Irigoin and R. Triolet. Supernode Partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages,* pp. 319–329, Jan. 1988.

[11] P. Kjeldsberg, F. Catthoor, and E. Aas. Automated data dependency size estimation with a partially fixed execution ordering. In *Proc. International Conference on Computer Aided Design (ICCAD 2000),* pp. 44–50, November 2000.

[12] P. Kjeldsberg, F. Catthoor, and E. Aas. Detection of Partially Simultaneously Alive Signals in Storage Requirement Estimation for Data Intensive Applications. In *Proc. 38th Design Automation Conference (DAC 2001),* pp. 365–370, June 2001.

[13] C. Lee and M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In Proc. *30th Annual International Symposium on Microarchitecture,* pp. 330–335, 1997.

[14] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. *Proc. 5th Workshop on Languages and Compilers for Parallel Computing,* August 1992.

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

[16] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation,* pp. 30–44, June 1991.

[17] M. Wolfe. *High Performance Compilers for Parallel Computing,* Addison-Wesley, 1996.

[18] Y. Zhao and S. Malik. Exact memory size estimation for array computation without loop unrolling. In *Proc. Design Automation Conference,* pp. 811–816, June 1999.

J. Ramanujam received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India in 1983, and his M.S. and Ph. D. degrees in computer science from The Ohio State University in 1987 and 1990 respectively. He is currently a Professor in the Department of Electrical and Computer Engineering at Louisiana State University. His research interests are in compilers for high-performance computer systems, embedded systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures and algorithms. He has published over nearly 120 papers in refereed journals and conferences in these areas in addition to several book chapters and a book. He received the National Science Foundation's Young Investigator Award in 1994. In addition, he has received the best paper awards at the 2003 International Conference on High Performance Computing (HiPC 2003) and the 2004 International Parallel and Distributed Processing Symposium (IPDPS 2004) for his work with others on compiler optimizations for quantum chemistry computations.

Jinpyo Hong received a bachelor and a master of engineering degree in Computer Engineering from Kyungpook National University in 1992 and 1994 respectively. For, the next three and half years, he worked at KEPRI (Korea Electrical Power Research Institute). He joined the graduate program in Electrical and Computer Engineering at Louisiana State University (LSU) in the Fall of 1997 and received his PhD degree in Electrical Engineering from LSU in August 2002. He was a research scientist at LSU between August 2002 and August 2004, where he was supported in part by a post-doctoral research fellowship from the National Science Foundation. Since August 2004, he has been at the University of Memphis, where he is an assistant professor of electrical and computer engineering. His research interests are in the areas of embedded systems, compiler optimizations, design automation, and special-purpose architectures.

Mahmut Kandemir received his M.S. in Control and Computer Engineering at Istanbul Technical University, Turkey in 1992 and his Ph.D. degree in Electrical Engineering and Computer Science from Syracuse University in 1999. He has been on the Penn State faculty since then, where he is now an associate professor. His research interests include embedded systems, power-conscious software development, optimizing and parallelizing compilers, input/output systems, and large-scale application analysis. He is a member of the Embedded and Mobile Computing Center at Penn State and actively collaborates with other faculty in this group. His most recent work focuses on design and implementation of energy-aware compiler optimizations and a heap-compressing garbage collector. His current work on energy-aware computation and embedded systems is supported by an NSF CAREER award and a DARPA/GSRC grant. In addition, he works on on optimizing compilers and in the area of large-scale data management concentrating on developing strategies for efficient input/output (I/O) on parallel systems.

Ashish Narayan received a Bachelor of Engineering degree from Sri Jayachamarajendra College of Engineering, Mysore, India in Electronics and Communications Engineering in December 1986. After working in India for several years, he came to Louisiana State University to attend graduate school, where he received a Master of Science in Electrical Engineering degree in May 1994. Since then, he has been working in the computer industry in the New Jersey area. His research interests include optimizing compilers and computer architecture.