Storage Requirement Estimation for Data Intensive Applications with Partially Fixed Execution Ordering

Per Gunnar Kjeldsberg
Norwegian University of Science and Technology
Trondheim, NORWAY

In this paper, we propose a novel storage requirement estimation methodology for use in the early system design phases when the data transfer ordering is only partly fixed. Using a representative application demonstrator, we show how our technique can effectively guide the designer to achieve a transformed specification with low storage requirement.

**MOTIVATION AND CONTEXT**
For data dominated HW/SW systems:
- Data transfer and storage determine cost and performance parameters.
- Must be main focus of the designer to achieve cost-optimized end product [Catthoor98].

![Data Path](Image640x37 to 712x58)

**DATA PATH**
- At system level no detailed storage requirement information is available.
- Estimation techniques are essential.
- High-level description characterized by large multi-dimensional loop nests and arrays.

![Memory Hierarchy](Image724x37 to 813x53)

**MEMORY HIERARCHY**
- Accurate estimates must take in-place mapping possibilities into account.
- Mainly decided by the ordering of the loops surrounding the arrays.
- Design decisions gradually fix this execution ordering.
- Estimates of upper and lower bounds on storage requirement needed at each step, given the partially fixed execution ordering.

**ESTIMATION METHODOLOGY**
- Previous work either assumes a fully fixed ordering, e.g. [Zha099], or does not take it into account at all [Balasa95].
- Generate data-flow graph reflecting data dependencies in the application code
- Estimate upper and lower bounds on dependency sizes per signal based on partly fixed execution ordering
- Traverse the data-flow graph to find the total size of simultaneously alive dependencies
- Bounds on the application's storage requirement

**Figure 1: Data dominated embedded system**

**Figure 2: Code example (MPEG-4 motion estimation kernel)**

```
for (y_s=0; y_s<=31; y_s++) {
    for (x_s=0; x_s<=31; x_s++) {
        for (y_p=0; y_p<=15; y_p++) {
            for (x_p=0; x_p<=31; x_p++) {
                if ((x_p == 0) & (y_p != 0)) sad[y_s][x_s][y_p][x_p] =
                else if ((x_p == 0) & (y_p == 0)) sad[y_s][x_s][y_p][x_p] =
                else if ((x_p == 0) & (y_p == 15)) sad[y_s][x_s][y_p][x_p] =
                else if ((x_p == 31) & (y_p != 0)) sad[y_s][x_s][y_p][x_p] =
                else if ((x_p == 31) & (y_p == 0)) sad[y_s][x_s][y_p][x_p] =
                else if ((x_p == 31) & (y_p == 15)) sad[y_s][x_s][y_p][x_p] =
                for (j=0 to 5) {
                    for (i=0 to 5) {
                        if (j>=2) & (i>=1)
                            B[i][j][k] = f( curr[y_p][x_p], prev[y_s+y_p][x_s+x_p] );
                        else if (j<2) & (i>=1)
                            B[i][j][k] = g(A[i-1][j-2][k] );
                        else if (j>2) & (i<2)
                            B[i][j][k] = g(A[i][j-2][k] );
                        else if (j<2) & (i<2)
                            B[i][j][k] = g(A[i-1][j][k] );
                    }
                }
            }
        }
    }
}
```

**Figure 3: Iteration space and production of array elements**

**Dependency Part (DP):**
Array elements produced by one instruction and read by another instruction, see Figure 4.

**Dependency Vector (DV):**
Vector in iteration space between two depending array elements. There is a DV between (i,j,k)-points (0,0,0) and (1,2,0), see Figure 5.

**Figure 4: Dependency Part**

**Figure 5: Dependency Vector and Dependency Vector Polytope**

**Bounds on the application's storage requirement with y_p outermost.**
Better ordering with y_s outermost was found.

```
for for (x_p=0; x_p<=31; x_p++) {
    A[i][j][k] = f( input );
    for k=0 to 2 {
        for j=0 to 5 {
            if (j>=2) & (i>=1)
                B[i][j][k] = f( curr[y_p][x_p], prev[y_s+y_p][x_s+x_p] );
            else if (j<2) & (i>=1)
                B[i][j][k] = g(A[i-1][j-2][k] );
            else if (j>2) & (i<2)
                B[i][j][k] = g(A[i][j-2][k] );
            else if (j<2) & (i<2)
                B[i][j][k] = g(A[i-1][j][k] );
        }
    }
}
```

**Figure 6: Spanning Dimension i fixed second outermost**

**Figure 7: Nonspanning Dimension k fixed outermost**

**Estimate with SD j innermost:**
- DP reduced → UB=18. DVP extended → LB=6. See Figure 6.

**Estimate with ND k outermost:**
- DP reduced → UB=12. DVP unchanged → LB=5. See Figure 7.

**Figure 8: Spanning Dimension j fixed second outermost**

**Figure 9: Total storage requirement a) No ordering, b) y_p outermost, c) y_s outermost, d) x_s second outermost, e) y_p third outermost and x_p fourth outermost**

**Figure 10:** Total storage requirement when the data transfer ordering is only partly fixed. Using a representative application demonstrator, we show how our technique can effectively guide the designer to achieve a transformed specification with low storage requirement.

**Figure 11:** Data dominated embedded system.

**Figure 12:** Code example (MPEG-4 motion estimation kernel).

**Figure 13:** Iteration space and production of array elements.

**Figure 14:** Spanning/Nonspanning Dimensions (SD/ND): The iteration space dimensions that are/are not a part of the DVP. In Figure 5: SD={i,j}, ND={k}.

**Figure 15:** Estimate with no execution ordering fixed:
Upper Bound (UB) = Size (DP) - Overlap = 36
Lower Bound (LB) = Size (DVP) - Overlap = 5

**Figure 16:** Dependency Vector Polytope (DVP):
- Size of simultaneously alive dependencies
- Intersection with the DP, see Figure 5.

**Figure 17:** Nonspanning Dimension k fixed outermost

**Figure 18:** Spanning Dimension j fixed second outermost

**Figure 19:** Upper Bound

**Figure 20:** Lower Bound

**MPEG-4 MOTION ESTIMATION KERNEL**
- Methodology employed during the early loop transformation design phase of an MPEG-4 motion estimation kernel, see Figure 2.
- Focus on the two-dimensional cur[y_p][x_p] and four-dimensional sad[y_s][x_s][y_p][x_p] arrays.
- The curr array has smallest storage requirement with y_p outermost.
- Estimates showed large penalty on sad array and total storage requirement, see Figure 9.
- Better ordering with y_s outermost was found.