DESIGNER'S NOTEBOOK

TEXAS INSTRUMENTS

Loop Partitioning on the 'C6x

Contributed by Richard Scales

Design Problem

The C6x code generation tools go through 5 basic phases when scheduling a loop as follows:

1. **Front-end C optimizations (C compiler only).**
2. **Instruction Selection (C compiler only).**
3. **Partitioning (C compiler and Assembly Optimizer).**
4. **Instruction Scheduling (C compiler and Assembly Optimizer).**
5. **Register Allocation (C compiler and Assembly Optimizer).**

The first three phases can have a direct impact on the 4th and most important stage, instruction scheduling. At each of the first three stages there are a set of heuristics which try to make intelligent decisions where multiple options exist. This designer notebook page focuses on the third stage, partitioning, as this can have great impact on the performance achieved with the tools.

Sometimes non-optimal partitioning can be the limiting factor in attaining the highest possible performance. Once the instructions are chosen in phase 2, the compiler and/or assembly optimizer must decide which instructions to execute on the A side and which to execute on the B side in phase 3. This can have a direct affect on res MII since there is only one cross path from A to B and one cross path from B to A available on any given cycle. If the partitioning is poor, either the number of cross paths or the number of functional units on a particular side can become a limiting factor in res MII.

Solution

The following code development flow is recommended to achieve the highest performance on loops:

1. Compile native C code
2. Add const declarations and loop count information.
3. Optimize C code using intrinsics and other methods.
4. Write linear assembly.
5. Add partitioning information to the linear assembly.

The fifth stage, partitioning, is necessary when optimal partitioning is not achieved with the Compiler or Assembly Optimizer.

Example 1 shows example feedback obtained from the Compiler

and Assembly Optimizer when using the -mw option. This information is valuable for pointing out potential problems with partitioning. Notice that the unpartitioned resource bound on the loop iteration interval is 3 but after partitioning it is 4. We can see below that this is due to 4 X cross paths on the A side and that there are 10 non-M unit instructions on the A side. Each of these force the minimum iteration interval to be at least 4.

```
;*----------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Loop Carried Dependency Bound : 3
;*      Unpartitioned Resource Bound  : 3
;*      Partitioned Resource Bound(*) : 4
;*      Resource Partition:
;*                                  A-side   B-side
;*      .L units                      0        0
;*      .S units                      2        2
;*      .D units                      2        2
;*      .M units                      2        2
;*      .X cross paths                4*       3
;*      .T address paths              2        2
;*      Long read  paths              1        0
;*      Long write paths              0        0
;*      Logical  ops (.LS)            4        1  (.L or .S)
;*      Addition ops (.LSD)           2        1  (.L or .S or .D)
;*      Bound(.L .S .LS)              3        2
;*      Bound(.L .S .D .LS .LSD)      4*       2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 4 iterations in parallel
;*      Done
;*----------------------------------------------------------*
```

Example 1. Example Feedback

By passing partitioning information to the Assembly Optimizer, it is possible to improve the loop minimum iteration interval to 3 even after partitioning. Example 2 shows the linear assembly for the feedback in Example 1. Notice the functional units specified in boldface. These pass enough information to the tools to improve the partitioning between the A and B sides of the loop.

```
_iir  .cproc   cptr0,sptr0
      .reg cptr1, s01, s10, s23, c10, c32, s10_s, s10_t
      .reg p0, p1, p2, p3, s23_s, s1, t, x, mask, sptr1
      .reg s10p, ctr
      MV        cptr0,cptr1
      MV        sptr0,sptr1
      MVK       50,ctr      ; setup loop counter
LOOP:   .trip 50
      LDW .D1T1 *cptr0,c32  ; CoefAddr[3] & CoefAddr[2]
      LDW .D2T2 *cptr1,c10  ; CoefAddr[1] & CoefAddr[0]
      LDW .D1T2 *sptr0,s10  ; StateAddr[1] & StateAddr[0]
      MV        s10,s10p    ; save StateAddr[1] & StateAddr[0]
      MPY .M1   c32,s10,p2  ; CoefAddr[2] * StateAddr[0]
      MPYH      c32,s10,p3  ; CoefAddr[3] * StateAddr[1]
      ADD       p2,p3,s23   ; CA[2] * SA[0] + CA[3] * SA[1]
      SHR       s23,15,s23_s; (CA[2]*SA[0] + CA[3]* SA[1])>>15
      ADD .2    s23_s,x,t   ; t=x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
      AND       t,mask,t    ; clear upper 16 bits
      MPY       c10,s10,p0  ; CoefAddr[0] * StateAddr[0]
      MPYH      c10,s10,p1  ; CoefAddr[1] * StateAddr[1]
      ADD       p0,p1,s10_t ; CA[0] * SA[0] + CA[1] * SA[1]
      SHR       s10_t,15,s10_s ; (CA[0]*SA[0] + CA[1]*SA[1])>>15
```

```
      ADD       s10_s,t,x ; x = t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)
      SHL       s10p,16,s1     ; StateAddr[1] = StateAddr[0]
      OR        t,s1,s01       ; StateAddr[0] = t
      STW .D1   s01,*sptr1     ; store StateAddr[1]& StateAddr[0]
[ctr] ADD         -1,ctr,ctr     ; dec outer lp cntr
[ctr] B           LOOP           ; Branch outer loop
        .endproc
```

Example 2. Linear Assembly for IIR Filter

Example 3 shows the improved result. Now the minimum iteration interval is 3 even after partitioning and a schedule with ii=3 is found.

```
;*----------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Loop Carried Dependency Bound : 3
;*      Unpartitioned Resource Bound  : 3
;*      Partitioned Resource Bound(*) : 3
;*      Resource Partition:
;*                                  A-side   B-side
;*      .L units                      0        0
;*      .S units                      2        2
;*      .D units                      3*       1
;*      .M units                      2        2
;*      .X cross paths                2        1
;*      .T address paths              1        3*
;*      Long read  paths              0        1
;*      Long write paths              0        0
;*      Logical  ops (.LS)            0        3  (.L or .S)
;*      Addition ops (.LSD)           2        3  (.L or .S or .D)
;*      Bound(.L .S .LS)              1        3*
;*      Bound(.L .S .D .LS .LSD)      3*       3*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 5 iterations in parallel
;*      Done
;*----------------------------------------------------------*
```

Example 3. Feedback after Partitioning

The goal in defining functional units and/or sides, is to split the loop into two halves (A and B) with minimal cross paths and minimal effect to the scheduling. When partitioning a loop in Linear Assembly try the following:

1. Minimize the number of cross paths. This usually involves looking at the dependency graph of the loop and splitting it such that there are the fewest number of paths crossing to the opposite side. Figure 1 shows a split where only one cross path is required. Keep in mind that dependencies due to conditional registers do not require a cross path (i.e. an instruction on the A side which is conditional on a B register does not use the cross path).

2. Choose a fairly even number of instructions for each side.

3. Force even numbers of certain functional units on each side. Figure 1 shows that even though there are 4 instructions that require a .D unit, they can be split on opposite sides to allow for an iteration interval of 2. The same is true of the three multiplies, rather than putting all 3 on the same side, one is put on the opposite side.

4. Force even numbers of instructions which write to a

conditional value on each side. Since there are a more limited number of conditional registers (there are 5 as opposed to the full 32 available for other source operands), it is easier to register allocate multiple conditional registers if they are split evenly between the two sides. If you have an uneven number, put more on the B side since there are 3 condition registers on this side and only 2 on the A side.

5. Use the T1 and T2 path directives to force the result of Loads and the source of Stores to a particular side (it can be different than the side the D unit is on). Refer to the Memory Banks section in the Assembly Optimizations chapter of the Programmer's Guide for more detailed information and examples.

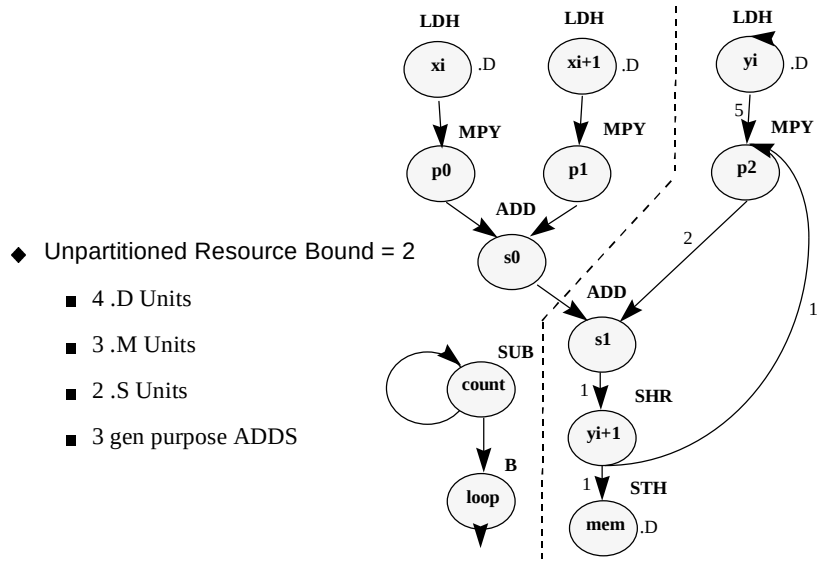

Figure 1. Splitting a Dependency Graph