Application Brief
SPRA534

Writing Interruptible Looped Code for the TMS320C6x DSP

Jackie Brenner *DSP Applications*

Abstract

Digital signal processing algorithms are loop intensive by nature, which presents a set of choices for the programmer working with the Texas Instruments (TI™) TMS320C6x digital signal processor (DSP). Loops are implemented on the C6x with a branch instruction. To maintain the determinism of operations within the C6x pipeline, a branch and its five delay slots are non-interruptible. The C6x code generation tools provide a high degree of flexibility for interruptibility. This application brief illustrates this flexibility and examines the code generated by various interruptibility strategies.

Contents

Problem

A requirement in any real-time system is the ability to respond to an interrupt. An interrupt is an asynchronous event that requires the CPU's attention and service. The flow of the main program is stopped to service the interrupt. After the interrupt is serviced, the main program resumes execution from the point it left off. DSPs compute algorithms quickly within the time period allowed by these asynchronous events in the real-time system. DSP algorithms are loop intensive by nature, which presents a set of choices for the C6x programmer.

Loops are implemented on the C6x with a branch instruction. On the C6x, a branch instruction executes in a single cycle but the effect of the branch is delayed by five cycles. To maintain the determinism of operations within the C6x pipeline, interrupts are disabled during a branch and these five cycles, or delay slots. All loops shorter than six cycles always have a pending branch. Therefore, all loops smaller than six cycles are non-interruptible.

This may or may not cause a problem in your system. Assume that you have a single cycle loop that is performed 100 times. As long as your interrupt threshold is longer than 500 ns (5 ns per cycle × 100 cycles), the loop can remain non-interruptible. If the interrupt threshold is less than 500 ns, you must increase the iteration interval (or the number of cycles required to do one instance of the loop) to six or greater to allow interrupts.

Solution

The C6x code generation tools provide a high degree of flexibility for interruptibility. The compiler option –mi*n* specifies an optional interrupt threshold value, *n*. The threshold value specifies the maximum number of clock cycles that the compiler can disable interrupts. When using the –mi*n* option, the compiler and assembly optimizer analyze both the loop structure and the number of times the loop will be iterated to determine the maximum number of cycles it will take to execute the loop. If the tools can determine that the maximum number of cycles is less than the threshold value, the compiler/assembly optimizer will create non-interruptible code. Otherwise, the tools generate interruptible looped code that will, in most cases, not degrade the performance of that loop.

The compiler command line option –mi*n* can be used for an entire module. In addition, a pragma can be used to specify the threshold on a function-by-function basis. This pragma is of the form:

#pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);

The #pragma overrides the –mi*n* command line option. If a threshold of less than 0 is specified, it is assumed that the function will never be interrupted.

Let us use an example to examine three cases: 1) code is never interruptible (*n* is infinity), 2) the code is always interruptible (*n* is 1) and 3) we give a specific interrupt threshold.

We use the following C /linear assembly code and examine the assembly code generated to see the effect of the interrupt threshold.

```
/* Prototype */
short DotP(short *m, short *n, short count);

/* Declarations */
short a[40] =
{40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18
,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
short x[40] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35, 36,37,38,39,40};
short y = 0;

/* Main Code */
main()
{
  y = DotP(a, x, 40);
}
```

This C file declares data arrays and makes a call to the function DotP that we have implemented in linear assembly. Note that it is our intention to run this procedure 40 times but we do not specify a trip count range in the linear assembly procedure below.

A trip count range can be used to define both the minimum and maximum number of times we expect the loop to run. The minimum input value for the trip count allows us to determine whether we can use software pipelining as a method to implement the loop. The maximum trip count value helps determine whether we can implement a non-interruptible loop given a particular interrupt threshold.

Another piece of information we can specify is a trip count factor, which indicates that the trip count is a multiple of some known number. The compiler can use the trip count factor along with the trip count range to unroll the loop to improve performance.

Software pipelining is a technique that takes advantage of the parallelism in the C6x's architecture by scheduling loop instructions so that we are working on different iterations of the loop at the same time.

```
             .title  "dotp_nt.sa"
             .def    _DotP
             .sect  "code"

_DotP:       .cproc      p_m, p_n, count
             .reg     m, n, prod, sum

               zero    sum
loop:
                ldh   *p_m++, m
                ldh   *p_n++, n
                mpy   m, n, prod
                add   prod, sum, sum

   [count]      sub   count, 1, count
   [count]      b     loop

                .return      sum
                .endproc
```

Non-Interruptible Code

We compile the above code with the following options: –gs –o2 –k –mw –mt –mi. The –g option enables symbolic debug, –s is for interlisting C and assembly language statements, and –o2 is the level of the optimizer that enables software pipelining. The –k option keeps the assembly language file, –mw gives us a report on how well the compiler is able to implement our loop, and –mt indicates we assume no aliasing (aliasing refers to multiple pointers pointing to the same object).

In this command line we did not specify a value for *n* with the –mi option. Therefore, the code generator will create code that has a threshold equal to infinity. If we do not use the –mi option, the default behavior of the compiler is enabled. This default behavior disables interrupts around all loops.

Here is an excerpt from the assembly code that is generated. We focus our attention on the loop kernel.

```
;******************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Thu Mar 11 07:27:24 1999
;******************************************************************

;******************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : Infinite
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;*
;******************************************************************
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1*
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        1   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1*
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
```

```
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 8 iterations in parallel
;*      Done
;*      Speculative Load Threshold  : 14
;*      Collapsed Epilog Stages     : 7
;*      Prolog not removed          : Ran out of functional units
;*      Collapsed Prolog Stages     : 0

; PIPED LOOP PROLOG

           LDH     .D2T2   *B5++,B4          ; |14|
||         LDH     .D1T1   *A3++,A0          ; |15|

   [ B0]   SUB     .L2     B0,0x1,B0         ; |19|
||         LDH     .D2T2   *B5++,B4          ; @|14|
||         LDH     .D1T1   *A3++,A0          ; @|15|

   [ B0]   B       .S2     loop              ; |20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @|19|
||         LDH     .D2T2   *B5++,B4          ; @@|14|
||         LDH     .D1T1   *A3++,A0          ; @@|15|

   [ B0]   B       .S2     loop              ; @|20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@|14|
||         LDH     .D1T1   *A3++,A0          ; @@@|15|

   [ B0]   B       .S2     loop              ; @@|20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@|14|
||         LDH     .D1T1   *A3++,A0          ; @@@@|15|

           MPY     .M1X    B4,A0,A4          ; |16|
|| [ B0]   B       .S2     loop              ; @@@|20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@@@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@@|14|
||         LDH     .D1T1   *A3++,A0          ; @@@@@|15|

           MPY     .M1X    B4,A0,A4          ; @|16|
|| [ B0]   B       .S2     loop              ; @@@@|20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@@@@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@@@|14|
||         LDH     .D1T1   *A3++,A0          ; @@@@@@|15|

loop:    ; PIPED LOOP KERNEL

   [ A1]   SUB     .S1     A1,1,A1           ;
||         ADD     .L1     A4,A5,A5          ; |17|
||         MPY     .M1X    B4,A0,A4          ; @@|16|
|| [ B0]   B       .S2     loop              ; @@@@@|20|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@@@@@|19|
|| [ A1]   LDH     .D2T2   *B5++,B4          ; @@@@@@@|14|
|| [ A1]   LDH     .D1T1   *A3++,A0          ; @@@@@@@|15|
```

Note we have created a single cycle loop with all instructions included in a pending branch and therefore non-interruptible. This code also illustrates the software pipelining technique. We perform all instructions required by the loop in parallel. Instructions that execute in parallel are also known as *execute packets*. The instructions required by the loop include the loading of two data values, multiplying two values, adding the result of the multiply to a summing register, decrementing a loop counter, and branching based on the value of that loop counter. But we are working on different iterations of the loop at the same time, as shown by the @ in the comment field. While we are doing our first add (no @), we are doing our third multiply, our sixth branch, our seventh subtraction of the loop counter, and our eighth load of the two data values.

Multiple Assignment

The software pipelining technique makes use of a concept called multiple assignment of registers. We mentioned above that we are working on different iterations of the loop in parallel. We are doing a load every cycle and a multiply every cycle in our loop above. However, our multiply instruction uses values that were previously loaded. In the case above, we multiply values that were loaded into registers B4 and A0 five cycles prior to the current cycle.

This has implications for interruptibility. All instructions that begin executing before an interrupt is taken will complete. If an interrupt occurs between the first loop iteration load and the first loop iteration multiply, we will get an invalid result. The reason for this is that the data load for the fifth loop iteration completes before we start the multiply for the first loop iteration. Therefore, incorrect data inputs are provided to the multiplier for the first four loop iterations.

To prevent an invalid result, the compiler by default turns off interrupts prior to entering a software pipelined loop and re-enables them after exiting a software pipelined loop whenever multiple assignment is utilized. (Please see the*TMS320C62x/C67x Programmer's Guide* for more information on single and multiple assignment.)

Code That is Always Interruptible

In our second example case we compile the same code but add the value of 1 to the –mi option (–mi1). This says we always want the code to be interruptible.

The following code is generated:

```
;*******************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Thu Mar 11 08:07:40 1999
;*******************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 1
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;*
;*********************************************************************

;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1*
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        1   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1*
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Loop is Interruptible
;*      Speculative Load Threshold  : 2
;*      Collapsed Epilog Stages     : 1
;*      Collapsed Prolog Stages     : 1
;*
;*---------------------------------------------------------------------

loop:    ; PIPED LOOP KERNEL

   [ B0]   B       .S2     loop              ; |19|
           NOP             2
           MPY     .M1X    B4,A0,A4          ; |15|

   [ A1]   LDH     .D2T2   *B5++,B4          ; @|13|
|| [ A1]   LDH     .D1T1   *A3++,A0          ; @|14|

   [ A2]   SUB     .D1     A2,1,A2           ;
|| [ A1]   SUB     .L1     A1,1,A1           ;
|| [!A2]   ADD     .S1     A4,A5,A5          ; |16|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @|18|
```

This creates a loop kernel that has six execute packets and is therefore interruptible.

Note also that we no longer have multiple assignment for registers B4 and A0. This code obeys single assignment. There is no pending next iteration load that occurs before the multiply happens. Now if there is an interrupt between the load and the multiply, the result will be correct because the completed load is in the same loop iteration as the multiply.

Interrupt Threshold

For our third example case, let us specify an interrupt threshold of 100 with a –mi100. We still do not modify the linear assembly to specify a trip count range or trip count factor.

Here is an excerpt from the output of the assembly optimizer:

```
;*******************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Thu Mar 11 09:03:57 1999
;*******************************************************************

;*******************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 100
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;*
;****************************************************************
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1*
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        1   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1*
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*

;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Loop is Interruptible
;*      Speculative Load Threshold  : 2
;*      Collapsed Epilog Stages     : 1
;*      Collapsed Prolog Stages     : 1
;*
;*

loop:    ; PIPED LOOP KERNEL

   [ B0]   B       .S2     loop              ; |19|
           NOP             2
           MPY     .M1X    B4,A0,A4          ; |15|

   [ A1]   LDH     .D2T2   *B5++,B4          ; @|13|
|| [ A1]   LDH     .D1T1   *A3++,A0          ; @|14|

   [ A2]   SUB     .D1     A2,1,A2           ;
|| [ A1]   SUB     .L1     A1,1,A1           ;
|| [!A2]   ADD     .S1     A4,A5,A5          ; |16|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @|18|
```

Note that this is identical to the –mi1 case. Because we did not specify a maximum trip count or trip count factor, the assembly optimizer creates code that is interruptible, with an iteration interval greater than or equal to six, and obeys single assignment.

Maximum Trip Count

Now let us modify the linear assembly to specify both a minimum and a maximum trip count and a trip count factor with the *.trip* directive.

```
              .title "dotp_ldh.sa"
              .def    _DotP
              .sect "code"

_DotP         .cproc   p_m, p_n, count
              .reg     m, n, prod, sum

              zero    sum

loop:         .trip   8, 40, 2 ; min. is 8, max. is 40, loop count is
                                ; always a multiple of 2
              ldh  *p_m++, m
              ldh  *p_n++, n
              mpy   m, n, prod
              add   prod, sum, sum
   [count]    sub   count, 1, count
   [count]    b     loop

             .return  sum
             .endproc
```

Now let us look at the output of the assembly optimizer when compiled with the –gs –o2 –k –mw -mt –mi100 options:

```
;**********************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Mon Mar 29 12:22:43 1999
;**********************************************************************
;* GLOBAL FILE PARAMETERS
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 100
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;*
;**********************************************************************
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 8
;*      Known Maximum Trip Count         : 40
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1*
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        1   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1*
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 8 iterations in parallel
;*      Done
;*
;*      Speculative Load Threshold  : 14
;*      Collapsed Epilog Stages     : 7
;*
;*      Prolog not removed          : Ran out of functional units
;*      Collapsed Prolog Stages     : 0
;*

; loop:           .trip   8,40,2

           LDH     .D2T2   *B5++,B4          ; |13|
||         LDH     .D1T1   *A3++,A0          ; |14|

           LDH     .D2T2   *B5++,B4          ; @|13|
||         LDH     .D1T1   *A3++,A0          ; @|14|

   [ B0]   B       .S2     loop              ; |19|
||         LDH     .D2T2   *B5++,B4          ; @@|13|
||         LDH     .D1T1   *A3++,A0          ; @@|14|

   [ B0]   B       .S2     loop              ; @|19|
||         LDH     .D2T2   *B5++,B4          ; @@@|13|
||         LDH     .D1T1   *A3++,A0          ; @@@|14|

   [ B0]   B       .S2     loop              ; @@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@|13|
||         LDH     .D1T1   *A3++,A0          ; @@@@|14|

           MPY     .M1X    B4,A0,A4          ; |15|
|| [ B0]   B       .S2     loop              ; @@@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@@|13|
||         LDH     .D1T1   *A3++,A0          ; @@@@@|14|

           MPY     .M1X    B4,A0,A4          ; @|15|
|| [ B0]   B       .S2     loop              ; @@@@|19|
||         LDH     .D2T2   *B5++,B4          ; @@@@@@|13|
||         LDH     .D1T1   *A3++,A0          ; @@@@@@|14|

loop:    ; PIPED LOOP KERNEL

   [ A1]   SUB     .S1     A1,1,A1           ;
||         ADD     .L1     A4,A5,A5          ; |16|
||         MPY     .M1X    B4,A0,A4          ; @@|15|
|| [ B0]   B       .S2     loop              ; @@@@@|19|
|| [ B0]   SUB     .L2     B0,0x1,B0         ; @@@@@@|18|
|| [ A1]   LDH     .D2T2   *B5++,B4          ; @@@@@@@|13|
|| [ A1]   LDH     .D1T1   *A3++,A0          ; @@@@@@@|14|
```

Now we again have a single cycle loop because we specified a maximum trip count of 40, which is below the interrupt threshold set. We can generate this same single cycle loop as long as the interrupt threshold is equal to or greater than 41.

Trip Count Factor

Specifying a trip count range and trip count factor can improve the performance of the loop even when the interrupt threshold is less than the maximum trip count. We mentioned above that the compiler can use the trip count factor along with the trip count range to unroll the loop to improve performance.

Let us take another look at the *.trip* directive we used in our last code example:

```
.trip   8, 40, 2 ; min. is 8, max. is 40, loop count is
                 ; always a multiple of 2
```

Our minimum trip count is 8, our maximum trip count is 40, and our trip count factor is 2 (the loop count will always be a multiple of 2).

We now change our interrupt threshold value to be 20 cycles with the –mi20 compiler option.

Here is an excerpt from the output of the assembly optimizer:

```
;**********************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Mon Mar 29 12:18:19 1999
;**********************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 20
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;**********************************************************************
```

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 4
;*      Known Maximum Trip Count         : 20
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     2*       2*
;*      .M units                     1        1
;*      .X cross paths               1        1
;*      .T address paths             2*       2*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        2   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        0
;*      Bound(.L .S .D .LS .LSD)     2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Loop is Interruptible
;*      Speculative Load Threshold  : 4
;*      Collapsed Epilog Stages     : 1
;*      Collapsed Prolog Stages     : 1
;*
; loop:           .trip   8,40,2

loop:    ; PIPED LOOP KERNEL
   [ B0]   B       .S1     loop                ; |19|
           NOP             1
           MPY     .M2X    B7,A5,B8            ; |15|

           MPY     .M1X    B6,A0,A6            ; |15|
|| [ A1]   LDH     .D2T2   *B4++(4),B7         ; @|13|
|| [ A1]   LDH     .D1T1   *A4++(4),A5         ; @|14|

   [!A2]   ADD     .L2     B8,B5,B5            ; |16|
|| [ A1]   LDH     .D2T2   *-B4(2),B6          ; @|13|
|| [ A1]   LDH     .D1T1   *-A4(2),A0          ; @|14|

   [ A2]   SUB     .D1     A2,1,A2             ;
|| [ A1]   SUB     .L1     A1,2,A1             ;
|| [!A2]   ADD     .S1     A6,A3,A3            ; |16|
|| [ B0]   SUB     .L2     B0,0x2,B0           ; @|18|
```

The compiler has created code that is still interruptible but, by unrolling the loop once, we are able to calculate two values per loop iteration. This allows us to double the performance from our previous case of interruptible code, in which we did not specify a maximum trip count or trip count factor.

Coding in C

We have now illustrated three cases of the compiler/assembly optimizer, generating code that is never interruptible, always interruptible, and interruptible based on an interrupt threshold. We have shown how in linear assembly we can use a trip count range and trip count factor to improve performance. Can this be done in the C environment alone?

The C6x compiler utilizes a number of intrinsic operators. Intrinsics are used as functions and produce assembly language statements that are ordinarily inexpressible in C. C variables are used with these intrinsics just as they would with any normal function. Starting with the 3.0 release of the C6x code generation tools, the intrinsic *_nassert* can be used to tell the compiler the minimum and maximum trip counts as well as trip count factor. The *_nassert* statement itself generates no code; it is analogous to the *.trip* directive in linear assembly.

Let's modify our original C program to include the *_nassert* intrinsic with a minimum count of 8, a maximum count of 40, and a trip count factor of 2. Note also we are no longer making a call to the DotP linear assembly function but we define the DotP in C:

```
/* Main Code */
main()
{

y = DotP(a, x, 40);
}


short DotP(short *m, short *n, short count)
{  int i;
   int product;
   int sum = 0;

 _nassert(count >=8 && count <=40 && (count % 2) == 0);

   for (i=0; i < count; i++)
   {

    product = m[i] * n[i];
    sum += product;
   }
   return(sum);
}
```

Now let us look at the assembly output of the compiler when the above code is compiled with the –gs –o2 –k –mw –mt –mi100 options:

```
;**********************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Tue Mar 30 14:11:57 1999
;**********************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 100
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : Debug
;**********************************************************************
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 8
;*      Known Maximum Trip Count         : 40
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 1
;*      Partitioned Resource Bound(*)    : 1
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1*
;*      .D units                     1*       1*
;*      .M units                     1*       0
;*      .X cross paths               1*       0
;*      .T address paths             1*       1*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          1        1   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1*
;*      Bound(.L .S .D .LS .LSD)     1*       1*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 1  Schedule found with 8 iterations in parallel
;*      Done
;*
;*      Speculative Load Threshold  : 14
;*      Collapsed Epilog Stages     : 7
;*
;*      Prolog not removed          : Ran out of functional units
;*      Collapsed Prolog Stages     : 0
;*
```

```
; PIPED LOOP PROLOG

           LDH     .D1T1   *A0++,A3          ; |27|
||         LDH     .D2T2   *B5++,B4          ; |27|

           LDH     .D1T1   *A0++,A3          ; @|27|
||         LDH     .D2T2   *B5++,B4          ; @|27|

   [ B0]   B       .S2     L2                ; |28|
||         LDH     .D1T1   *A0++,A3          ; @@|27|
||         LDH     .D2T2   *B5++,B4          ; @@|27|

   [ B0]   B       .S2     L2                ; @|28|
||         LDH     .D1T1   *A0++,A3          ; @@@|27|
||         LDH     .D2T2   *B5++,B4          ; @@@|27|

   [ B0]   B       .S2     L2                ; @@|28|
||         LDH     .D1T1   *A0++,A3          ; @@@@|27|
||         LDH     .D2T2   *B5++,B4          ; @@@@|27|

           MPY     .M1X    B4,A3,A5          ; |27|
|| [ B0]   B       .S2     L2                ; @@@|28|
||         LDH     .D1T1   *A0++,A3          ; @@@@@|27|
||         LDH     .D2T2   *B5++,B4          ; @@@@@|27|

           MPY     .M1X    B4,A3,A5          ; @|27|
|| [ B0]   B       .S2     L2                ; @@@@|28|
||         LDH     .D1T1   *A0++,A3          ; @@@@@@|27|
||         LDH     .D2T2   *B5++,B4          ; @@@@@@|27|

; PIPED LOOP KERNEL

   [ A1]   SUB     .S1     A1,1,A1           ;
||         ADD     .L1     A5,A4,A4          ; |27|
||         MPY     .M1X    B4,A3,A5          ; @@|27|
|| [ B0]   B       .S2     L2                ; @@@@@|28|
|| [ B0]   SUB     .L2     B0,1,B0           ; @@@@@@|28|
|| [ A1]   LDH     .D1T1   *A0++,A3          ; @@@@@@@|27|
|| [ A1]   LDH     .D2T2   *B5++,B4          ; @@@@@@@|27|
```

The compiler generated a single cycle loop, just as we saw with the assembly optimizer. This time we remained entirely in the C environment with the addition of the _*nassert* intrinsic. A single cycle loop was possible because the specified maximum trip count (40) was below the interrupt threshold that was set.

Optimum Performance in C With Interruptibility

Let’s look at one last case where the interrupt threshold is less than the maximum specified trip count. In our linear assembly example we doubled the performance of an interruptible loop by specifying a trip count factor of 2. The trip count factor specifies that the loop counter is a multiple of the number provided. What if we know that the trip count will always be a multiple of 8? We can modify the trip count factor, which allows the C compiler to unroll the loop even further to obtain optimum performance while maintaining interruptibility.

Let’s modify the *_nassert* intrinsic in our C program to have a minimum count of 8, a maximum count of 40, and a trip count factor of 8:

```
/* Main Code */
main()
{

y = DotP(a, x, 40);
}


short DotP(short *m, short *n, short count)
{  int i;
   int product;
   int sum = 0;

 _nassert(count >=8 && count <=40 && (count % 8) == 0);

   for (i=0; i < count; i++)
   {

    product = m[i] * n[i];
    sum += product;
   }
   return(sum);
}
```

This time we compile the above code with –o3 level of optimization and do not keep track of debug or interlisting information. We also use the –mx option, which tells the compiler to spend more time to find an optimum solution. We also utilize an interrupt threshold of 20, which is less than our maximum trip count of 40. Our compiler command line options are now: –o3 –k –mx –mw –mt –mi20. The following loop kernel is generated by the compiler:

```
;**********************************************************************
;* TMS320C6x ANSI C Codegen
;*Version 3.00 *
;* Date/Time created: Mon Apr 05 09:18:51 1999
;**********************************************************************
;* GLOBAL FILE PARAMETERS
;*
;*   Architecture        : TMS320C6200
;*   Endian              : Little
;*   Interrupt Threshold : 20
;*   Memory Model        : Small
;*   Calls to RTS        : Near
;*   Pipelining          : Enabled
;*   Speculative Load    : Threshold = 0
;*   Epilog Collapsing   : Enabled
;*   Prolog Collapsing   : Enabled
;*   Redundant Loops     : Enabled
;*   Code Size Opt.      : Disabled
;*   Memory Aliases      : Presume not aliases (optimistic)
;*   Debug Info          : No Debug Info
;**********************************************************************
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop Unroll Multiple             : 3x
;*      Known Minimum Trip Count         : 3
;*      Known Maximum Trip Count         : 3
;*      Known Max Trip Count Factor      : 3
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 6
;*      Partitioned Resource Bound(*)    : 6
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     6*       6*
;*      .M units                     6*       6*
;*      .X cross paths               6*       6*
;*      .T address paths             6*       6*
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0   (.L or .S unit)
;*      Addition ops (.LSD)          6        7   (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        0
;*      Bound(.L .S .D .LS .LSD)     5        5
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Cannot allocate machine registers
;*                   Regs Live Always   :  7/8  (A/B-side)
;*                   Max Regs Live      : 14/15
;*                   Max Cond Regs Live :  0/1
;*         ii = 7  Cannot allocate machine registers
;*                   Regs Live Always   :  7/8  (A/B-side)
;*                   Max Regs Live      : 13/14
;*                   Max Cond Regs Live :  0/1
;*         ii = 8  Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Loop is Interruptible
;*      Epilog not removed          : Instructions share increment
;*      Speculative Load Threshold  : 24
;*      Collapsed Epilog Stages     : 0
;*
;*      Collapsed Prolog Stages     : 1
;*
```

```
; PIPED LOOP KERNEL

            MPY     .M2X    B11,A7,B11        ; |27|
||          MPYH    .M1X    B11,A7,A7         ; |27|
|| [!A1]    LDW     .D1T1   *+A13(4),A2       ; |27|
|| [!A1]    LDW     .D2T2   *+B6(4),B1        ; |27|

   [ B0]    SUB     .L2     B0,3,B0           ; |28|
||          MPY     .M2X    B13,A0,B13        ; |27|
||          MPYH    .M1X    B13,A0,A0         ; |27|

   [!A1]    ADD     .L2     B11,B4,B4         ; |27|
|| [!A1]    ADD     .L1     A7,A10,A10        ; |27|
||          MPY     .M2X    B12,A3,B12        ; |27|
||          MPYH    .M1X    B12,A3,A3         ; |27|
|| [ B0]    B       .S1     L2                ; |28|

   [!A1]    ADD     .L2     B13,B5,B5         ; |27|
|| [!A1]    ADD     .L1     A0,A9,A9          ; |27|
||          MPY     .M2X    B2,A6,B2          ; |27|
||          MPYH    .M1X    B2,A6,A6          ; |27|
||          LDW     .D1T1   *+A13(8),A7       ; @|27|
||          LDW     .D2T2   *+B6(8),B11       ; @|27|

   [!A1]    ADD     .L2     B12,B7,B7         ; |27|
|| [!A1]    ADD     .L1     A3,A8,A8          ; |27|
||          MPY     .M2X    B3,A4,B3          ; |27|
||          MPYH    .M1X    B3,A4,A4          ; |27|
||          LDW     .D1T1   *+A13(12),A0      ; @|27|
||          LDW     .D2T2   *+B6(12),B13      ; @|27|

   [!A1]    ADD     .L2     B2,B10,B10        ; |27|
|| [!A1]    ADD     .L1     A6,A12,A12        ; |27|
||          MPY     .M2X    B1,A2,B1          ; |27|
||          MPYH    .M1X    B1,A2,A2          ; |27|
||          LDW     .D1T1   *+A13(16),A3      ; @|27|
||          LDW     .D2T2   *+B6(16),B12      ; @|27|

   [!A1]    ADD     .L2     B3,B9,B9          ; |27|
|| [!A1]    ADD     .L1     A4,A11,A11        ; |27|
||          LDW     .D1T1   *+A13(20),A6      ; @|27|
||          LDW     .D2T2   *+B6(20),B2       ; @|27|

   [ A1]    SUB     .S1     A1,1,A1           ;
|| [!A1]    ADD     .L2     B1,B8,B8          ; |27|
|| [!A1]    ADD     .L1     A2,A5,A5          ; |27|
||          LDW     .D1T1   *++A13(24),A4     ; @|27|
||          LDW     .D2T2   *++B6(24),B3      ; @|27|
```

At the –o3 level of optimization, the compiler creates code that brings in two 16-bit values per load with an LDW instruction. It also uses the second multiplier on the C6x with an MPYH instruction (multiply the upper 16 bits of a register by the upper 16 bits of the second register). In addition, the compiler has unrolled the loop a total of three times, creating an eight-cycle loop in which 12 multiplies are executed per loop iteration. This results in no performance degradation from our single-cycle loop case but our code size has grown. In fact, the performance of our loop has increased at this level of optimization because we are averaging 1.5 multiplies per cycle instead of 1 multiply per cycle.

Conclusion

The C6x code generation tools provide a high degree of flexibility for interruptibility. We can specify an interrupt threshold globally through a compiler option or use a pragma to change interruptibility on a function-by-function basis. We can also use the flexibility of the tools to create interruptible code with no loss of performance. This application brief illustrates this flexibility and examines the code generated by various interruptibility strategies.

References

TMS320C6x Optimizing C Compiler User's Guide, Literature number SPRU187, Texas Instruments Inc, 1998.

TMS320C62x/C67x Programmer's Guide, Literature number SPRU198, Texas Instruments Inc., 1998.

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Email	sc-infomaster@ti.com

Europe, Middle East, and Africa

Phone	
Deutsch	+49-(0) 8161 80 3311
English	+44-(0) 1604 66 3399
Español	+34-(0) 90 23 54 0 28
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+44-(0) 1604 66 33 34
Email	epic@ti.com

Japan

Phone	
International	+81-3-3457-0972
Domestic	0120-81-0026
Fax	
International	+81-3-3457-1259
Domestic	0120-81-0036
Email	pic-japan@ti.com

Asia

Phone	
International	+886-2-23786800
Domestic	
Australia	1-800-881-011
TI Number	-800-800-1450
China	10810
TI Number	-800-800-1450
Hong Kong	800-96-1111
TI Number	-800-800-1450
India	000-117
TI Number	-800-800-1450
Indonesia	001-801-10
TI Number	-800-800-1450
Korea	080-551-2804
Malaysia	1-800-800-011
TI Number	-800-800-1450
New Zealand	000-911
TI Number	-800-800-1450
Philippines	105-11
TI Number	-800-800-1450
Singapore	800-0111-111
TI Number	-800-800-1450
Taiwan	080-006800
Thailand	0019-991-1111
TI Number	-800-800-1450
Fax	886-2-2378-6808
Email	tiasia@ti.com

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.