In just a few short phases, you can optimize C-coded reference modems to meet higher-density targets.

Optimizing Modems Using Code Composer Studio and TI Resources

By Ghassan Farah

For many of the general telephony stream-processing tasks, the C optimizer for the Texas Instruments TMS320C6000 DSP platform can yield higher densities with no assembly coding. Other technologies require a healthy dose of optimization to reach target densities.

You can take steps to optimize C-coded reference modems to meet higher-density targets. How high? C-baseline modems, for example, can soar from 6 per 200-MHz C6201 to 28 in four short project phases. A fifth project phase can take the number of channels to 48 per DSP.

In fact, for the MSP MEDIA Gateway line of DSP resource boards based on C6000 DSPs, Commetrex undertook the four phases, and the process worked. Our MSP-320 PCI board, with two C6201 DSPs and a quad E1/T1 network interface, needed 48 to 60 channels of processing from each DSP. For many of the general telephony stream-processing tasks, the C6000 C optimizer gave us the densities we needed with no assembly coding.

"Out-of-the-box" C-coded modems, which are a reference design and written for understandability rather than efficiency, might compile to, say, six simultaneous modems. You should be able to double that by guiding the modems through the Code Composer Studio (CCS) optimizer and by ensuring that your memory layout takes advantage of the C6000's on-chip RAM.

CCS includes an optimization tutorial that provides a recommended code development flow consisting of four phases (Figure 1). (A similar tutorial is in the *TMS320C6000 Programmer's Guide.)*

Phase 1 involves compiling and profiling your baseline C code. Before you begin any optimization effort, use the profiling tools to identify the performance-critical areas in your code.

Phase 2 involves compiling with the appropriate optimization options and analyzing the feedback provided by the compiler to improve the performance of your code.

Phase 3 is a critical phase during which you use a number of techniques to tune your C code for better performance.

Phase 4 is needed if the performance of certain areas of your code must be improved beyond the tuning phase. After yet another profile of the code, you can extract the performance-critical areas and rewrite them in linear assembly language.

THE FIRST THREE PHASES

Phase 1 establishes your baseline. You have a goal—for example, your system requirement might be a statistical mix of 48 modems on one

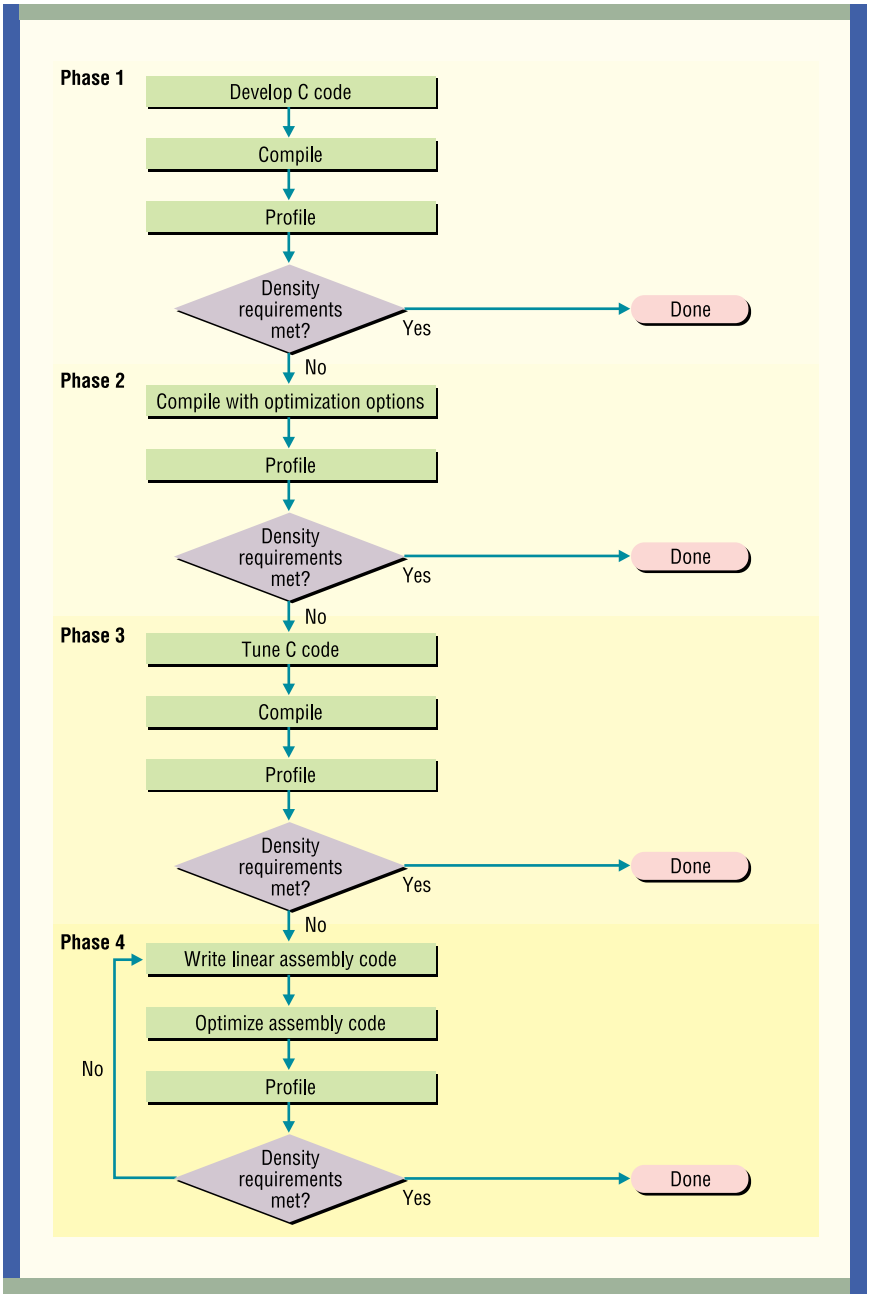

Figure 1. The Code Composer Studio optimization tutorial recommends a code development flow consisting of four phases. The first three phases focus on utilizing the optimization abilities of the TMS320C6000 compiler to achieve high code performance while maintaining the code in C. The last phase involves linear assembly coding of those portions of the code whose performance needs to be improved further. (This figure is based on the one on p. 1-4 of the* TMS320C6000 Programmer's Guide*).

200-MHz C6201. Always maintain the C-coded baseline as your reference code. Because it's often developed very straightforwardly, leaving it as a reference will be valuable if you have to diagnose a problem. Make your improvements there, then factor them into the optimized version to produce a bit-exact version.

Phase 2 involves compiling using the optimization options. The optimizer, combined with a judicious memory layout, can more than double the number of modems on one chip. Allocate a few weeks for the effort. But note that the optimizer is capable of "breaking" the modems in a few places, so you may have to modify some pieces of the C code. Also, you may find that some of the changes you make to the C code to improve the optimizer's results when using CCS 1.1 aren't required when using the release 1.2 optimizer; therefore move to 1.2 if you can.

In phase 3, you tune the C code. There are a number of techniques to refine your C code and greatly increase its efficiency. The goal is to allow the compiler to schedule as many instructions as possible in parallel, especially for MIPS-intensive loops, by providing information concerning the dependencies between instructions. You can use certain key words that give the compiler hints as it tries to determine dependencies.

Another useful technique in the tuning phase is to use intrinsics, which are special functions that map directly to C6000 assembly instructions. These functions are usually not easily expressed in C. They allow you to have more precise control over the selection of instructions by the compiler.

For example, some intrinsics operate on data stored in the low and high portions of a 32-bit register. Consequently, if you're operating on a stream of 16-bit values, you

Listing 1: Fixed-Point FIR Filter with Data Move

```
/*************************************************************
 * Routine Name: FIR_Filter_Shift
 *
 * Description:
 *    Performs fixed-point implementation of FIR filtering with
 *    data move.
 *
 *
 * Calling Sequence:
 *
 * short FIR_Filter_Shift(         short *taps,
 *                                 short *coefs,
 *                         unsigned short length,
 *                         unsigned short base)
 *    Where:
 *       taps   = pointer to filter taps delay line
 *       coefs  = pointer to filter coefficients, which are
 *                  stored in reverse order
 *       length = length of taps delay line
 *       base   = base of filter coefficients
 *
 * Returns:
 *    A filtered sample
 *************************************************************/

short FIR_Filter_Shift(            short *taps,
                                   short *coefs,
                           unsigned short length,
                           unsigned short base)
{
   int sum = 0;

   taps  += length - 1;

   while (length--)
   {
      sum  += *taps * *coefs++;
      *taps-- = *(taps-1);
   }

   /* round and remove base */
   return( ( sum + (1<<(base-1)) ) >> base );
}
```

can use word (32-bit) accesses to read and process two 16-bit values at a time.

Even though phases 2 and 3 may double the number of simultaneous instances of the code running on one chip, the modems are still coded in C that's easy to understand and maintain.

PHASE 4: CIRCULAR ADDRESSING

At this point, if your performance requirements are not yet met, you go on to phase 4: converting MIPS-intensive portions of the code into linear assembly code. This form of assembly code doesn't require that you provide functional unit selection, pipelining, parallelization, or register allocation; those tasks will still be performed by the compiler. It will, however, give you more control over the exact C6000 instructions to be used. You can also pass more useful information to the tools, such as which memory bank is to be used.

Modems use a number of delay lines for the different filters, resulting in MIPS-intensive memory shifting. You can avoid that by employing the circular addressing feature of the C6000 in your linear assembly code. It's not unreasonable to set a goal of doubling the number of modems from 12 to 24 in this step alone.

For the most part, a modem is a series of filters. Each filter is computed from a sequence of input data, or taps, and an equal number of coefficients. A multiply-accumulate operation is performed with each tap and a corresponding coefficient. After the computation, the taps are shifted to make room for the new input (Figure 2a). Circular addressing changes the starting point for the MAC cycle, eliminating the shifting altogether (Figure 2b).

Without hardware support for this operation, the C code for the iterative loop is of the form in Listing 1. The C6000 has hardware support for circular addressing, though. By setting the addressing mode register (AMR) appropriately, you can specify the general-purpose register or registers that will be used for circular addressing, as well as the size of the memory block that will be addressed circularly (Listing 2).

Just as using the optimizer has its challenges, so can adding circular addressing. You might find that you add circular addressing and then the optimizer breaks it. It turns out the optimizers in both CCS 1.1 and 1.2 don't take circular addressing into account. For example, the optimizer will often move an address from a register configured for circular

addressing to another register before performing address manipulations.

When using the optimizer with circular addressing, you might have to experiment with a number of alternative codings to arrive at a solution that the optimizer respects. (The new Code Composer Studio 2.0 from TI supports circular addressing directly from C code.)

You should see a significant improvement with circular addressing. Take our V.29 receiver (9,600 b/s) as an example: After the first three phases of our project, it consumed 222,188 cycles for each 10 ms of PCM data (80 samples). By modifying just the first two sections—the pulse-shaping and Hilbert filters—for circular addressing, we brought that down to 185,759 cycles. Changing the interpolating and baud timing recovery filters to the circular addressing mode reduced it to 155,677. Finally, changing the adaptive filtering and update routines shrank the cycle count down to 101,429—a reduction of better than 55 percent. (For a more in-depth discussion of circular addressing on the C6000, refer to the TI Application Report *Circular Buffering on TMS320C6000* [SPRA645.PDF].)

Since a V.17 receiver (14,400 b/s) is essentially the same code as the V.29 receiver but executes from different tables, these changes cause similar reductions to the V.17 receiver. However, we still need to optimize the Viterbi decoder. (Of the three common modems used to transfer fax-image data, only the V.17 modem uses Viterbi decoding.)

Trellis coding is a forward error correction scheme that reduces a modem's bit-error rate for a given amount of channel noise by adding certain redundant information to the channel. The information reduces the chance that noise will create data errors, in effect increasing the distance between code points. The Viterbi decoder decodes the Trellis sequence and determines the most likely set of transmitted points. However, it's expensive in terms of MIPS. C-coded Viterbi decoder alone took 140 percent of the cycles that the entire V.29 receiver took. In other words, the V.17 receiver was 2.4 times as expensive as the V.29.

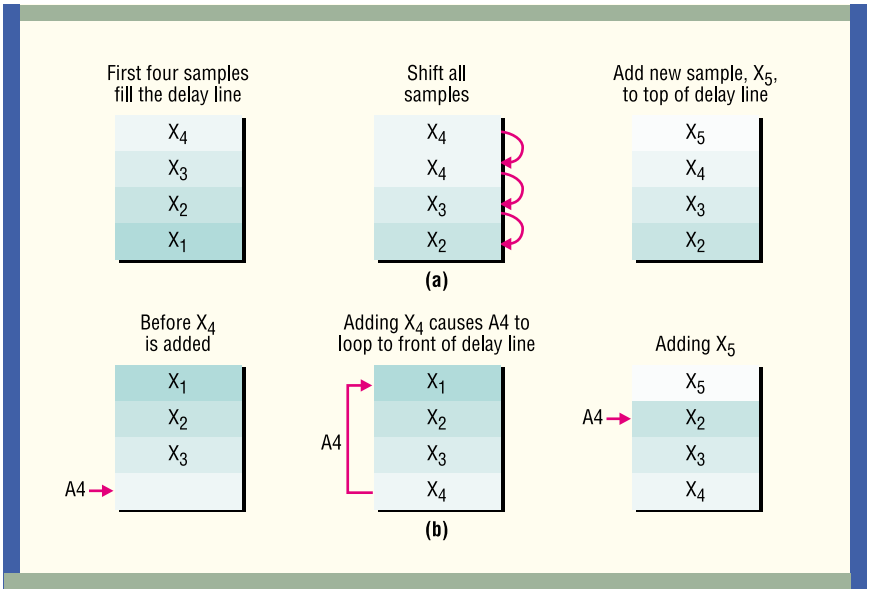

Figure 2. A new sample, x_5, is added to the delay line of a four-tap filter (x_1 is the oldest sample in time) using the sample-shifting method. Three shifts are needed before x_5 is placed at the top of the delay line (a). Using circular addressing, register A4 (set up to be used in circular mode) automatically wraps back to the beginning of the delay line after x_4 is added and the end of the delay line is reached. When x_5 is added, it overwrites the oldest available sample, x_1, thereby eliminating the shifting altogether (b).

Fortunately, help is available on the TI Web site (www.ti.com). When you download *Implementing V.32bis Viterbi Decoding on the TMS320C6200 DSP* (SPRA444.PDF), you'll find the decoder in very tight assembly code. You can't just drop it in, though. You'll have to adapt it to your environment.

To make the decoder reentrant, change global variables to per-channel contexts and watch for bugs. You should achieve spectacular results: A straight C-coded Viterbi consumes approximately 150,000 cycles for 80 samples. Substituting TI's assembly code takes that down to an incredible 8,000 cycles. Our V.29 receiver is now 101,429 cycles, and the V.17 receiver only 108,840—and we haven't begun to "vectorize."

Using a statistical mix of modems yields 28 simultaneous channels. In worst-case nonblocking terms, that's 18 simultaneous V.17 receivers. You should receive similar results for similar algorithms by using the optimizer and circular addressing.

BEYOND PHASE 4: 'VECTORIZE'

If you still haven't reached your performance requirements, you

Listing 2: FIR Filter Using Circular Addressing with Hardware Support

```
; Replacement for FIR_Filter_Shift
; PARAMETERS:
;    A4 - (in) *coefs
;    B4 - (in) *taps (base address of circular buffer)
;    A6 - (in) length (length of delay line)
;    B6 - (in) block_size
;    A8 - (in) write_offset (where the next value will be written)
;    B8 - (in) base

       .global _FIR_Filter

_FIR_Filter    .cproc  coef_block,taps,A6,B6,A8,base

               .reg    old_amr,ar,coef,sum1,dl,one,round,offs

       ; coef_block = coefs + ((length-1) * sizeof(short))
       SUB     A6,1,dl                    ; dl = length - 1
       SHL     dl,1,dl                    ; dl = (length-1) * sizeof(short)
       ADD     coef_block, dl,coef_block  ; coef_block = coefs + dl

       SHL     B6,16,B6            ; set block size of circular buffer
       SET     B6,8,8,B6           ; set B4 to circular mode
       MVC     AMR,old_amr         ; old_amr = AMR
       MVC     B6,AMR              ; AMR = addressing_mode

       ZERO    A1                  ; sum = 0

       ; ar + write_offset is where we will write the next sample
       ; advance to the most recent word (write_offset - 2)
       ; ar = taps + (write_offset - 2)
       SUB     A8,2,offs
       ADDAB   taps,offs,taps
       MV      A6,dl

startloop:     .trip   1
       LDH     *taps--,ar
       LDH     *coef_block--,coef
       MPY     ar,coef,sum1
       ADD     sum1,A1,A1
       SUB     dl,1,dl
 [dl]  B       startloop

       ; round and remove base
       ; sum = ( sum + (1<<(base-1)) ) >> base
       SUB     base,1,round
       MVK     1,one
       SHL     one,round,round
       ADD     round,A1,A1
       SHR     A1,base,A1

       MVC     old_amr,AMR         ; restore original addressing mode

       .return A1
       .endproc
```

might consider going on to phase 4: changing the flow of data through your code to reduce function calls and utilize more loops that can be optimized easily. For modems, one approach to accomplish that is to "vectorize" the algorithm's implementation.

The sample rate section of the receiver consists of the following components in series: the pulse-shaping filter, the Hilbert transformer, the demodulator, and the interpolator. Without vectorization, the sample rate section of the receiver processes one sample at a time, taking it through each successive section. Consequently, the overhead of calling each filter in the sample rate section is incurred 80 times for each 80-sample buffer. With vectorization, the sample rate section is called once for each 80-sample buffer. An input buffer of 80 samples is then passed to the pulse-shaping filter, which produces 80 samples to be passed to the Hilbert filter, which in turn produces 80 outputs, and so on. In the sample rate section, the number of function calls required to process 80 samples is reduced from 320 to just 4. In addition, processing the input buffer in a loop format as opposed to sample by sample allows the optimizer to do a better job of pipelining, significantly improving efficiency.

We haven't completed the vectorization phase of this project, but we will report the results on our Web site (www.commetrex.com) when we do. ◆

Ghassan Farah (Ghassan_Farah@commetrex.com) is manager, signal processing technologies, at Commetrex Corporation in Norcross, Ga. He has four years' experience in designing and implementing a variety of DSP algorithms. His technical interests include data and fax modems, telephony, speech coding, and signal classification.

TCP/IP *for* TI DSPs

Delivering precisely what you need.

Since 1994, Precise has been a leading supplier of embedded Internet protocols on Texas Instruments' TMS320™ DSP Family.

TMS320C3x™ 1994

TMS320C4x™ 1995

TMS320C62x™ 1998

TMS320C67x™ 1999

TMS320C54x™ 1999

PRECISE™ Software Technologies Inc.

Precise Software Technologies Inc.
301 Moodie Drive, Suite 308
Nepean, Ontario
Canada K2H 9C4
Sales: 800-265-9833
Phone: 613-596-2251
Fax: 613-596-6713
E-mail: info@psti.com

DSP

www.psti.com