D8.3

The Application of High-Level Languages to Single-Chip Digital Signal Processors

Ray Simar Jr. and Alan Davis
Texas Instruments, Inc.
P.O. Box 1443 M/S 701
Houston, Texas 77251-1443

Abstract

This paper discusses strategies leading to the efficient use of High-Level Languages (HLLs) on single-chip Digital Signal Processors (DSPs). The paper takes the reader from the specification of a particular algorithm, through stepwise refinements, to an optimized implementation for the DSP. For the purpose of illustration, Texas Instruments' high-performance floating-point DSP, the TMS320C30, and its optimizing C-compiler are used. Further discussion is given to the execution of general-purpose code where the TMS320C30 and its optimizing C-compiler combine to attain 10,987 Dhrystones/sec.

1 Introduction

The first programmable single-chip Digital Signal Processors (DSPs), such as the TMS32010, possessed a small set of general-purpose features and a small address range. The principle mode for programming these devices is with assembly languages targeted for the specific DSP. Recent years have seen DSPs attain a greater address range, 64 Mbytes in the case of the TMS320C30, and more general-purpose features. With these developments, one has to ask if the DSP-system designer might not be able to exploit High-Level Languages (HLLs) in much the same way as the general-purpose system designer. This paper will present several techniques of program development and refinement that allow the programmer not only to enjoy the ease of programming attributed to HLLs but also to achieve optimal or near-optimal performance.

1.1 TMS320C30 Features Supporting High-Level Languages

The formation of efficient assembly-language code from an algorithm specified using a HLL is inevitably dependent upon a successful marriage of silicon technology and compiler technology. The TMS320C30 ([1]) has features which are useful not only to the assembly-language programmer but also to the compiler writer. These features include a register based 32/40-bit CPU with eight data and eight address registers, single-cycle floating-point operations, on-chip memory for local and global variable storage, an on-chip program cache, separate program and data buses, flexible addressing modes, and orthogonal instruction classes including three-operand instructions.

1.2 Description of the TMS320C30 Optimizing C-Compiler

The optimizing C-compiler for the TMS320C30 was designed with two major goals in mind:

1. For general-purpose C code, produce compiled code that performs nearly as well as hand-coded assembly.
2. Provide a simple and accessable programming environment so that applications demanding higher performance can be implemented using assembly language for critical code and still use C for general-purpose code.

This philosophy relies heavily on the proven concept that an application spends 90% of its time in 10% of its code. The compiler is intended to do a good job with the remaining 90% of the code, and in all but the most demanding applications, can be used for the time-critical 10% as well.

Three distinct approaches were used to achieve the compiler design-goals. First, the compiler performs a number of built-in optimizations which can be thought of as the "engine" of the compiler, providing a given "horsepower". Second, the compiler is designed so that a knowledgeable programmer can help the compiler generate good code by writing the C program to take advantage of certain features of the C language. These features can be thought of as the compiler's "steering wheel" and "gas pedal." Finally, the compiler is designed so that if it is necessary to write part of the application in assembly language, it is easily done. One might say that it is easy to "open the hood."

We will use all of these approaches in our process of stepwise refinement. But first we will look at a few of the built-in optimizations the compiler performs. These optimizations include register tracking, jump optimizations, expression reordering, and dead-code removal.

1.2.1 Register Tracking

The compiler keeps track of the contents of registers so that it avoids reloading values if they are used again soon. For example, given the following code fragment:

```
int a = 10;
a + b;
```

The compiler, rather than generating

```
LDI  10,R0       will generate      LDI  10,R0
STI  R0,_a                          STI  R0,_a
LDI  _a,R1                          ADDI _b,R0
ADDI _b,R1
```

The compiler tracks variables, direct structure references, constants, and addresses of variables. The compiler also tracks register contents across branches.

1.2.2 Jump Optimizations

Often, the compiler generates branches to other branches. This happens when control structures are nested within each other. The compiler reduces branching chains to a single branch that jumps to the final destination. Therefore, rather than the compiler generating

```
L1:    ADDI RO,R1        the compiler    L1:    ADDI RO,R1
       JNZ  L2            generates             JNZ  L1
       LDI  _a,RO                               LDI  _a,RO
L2:    JMP  L1                                  JMP  L1
```

1.2.3 Expression Reordering

In many cases, the compiler generates better code for an expression by reordering the expression. For example, -(a + b) is treated as if it were the algebraically equivalent -a - b and the compiler, rather than generating

```
   LDI  _a,RO           will generate    NEGI _a,RO
   ADDI _b,RO                            SUBI _b,RO
   NEGI RO
```

The compiler performs dozens of optimizations that involve reordering or distributing operations. Such optimizations are performed for logical, pointer, relational, and assignment operations as well as the standard arithmetic ones.

1.2.4 Dead-Code Removal and Other Optimizations

Occasionally, a user writes a section of code that cannot be reached and so will never be executed. This section of code is referred to as *dead-code*. The compiler detects dead code and removes it.

The compiler performs numerous other optimizations to generate better code for various constructs. These are somewhat miscellaneous, but generally locally affect the code generated for a given operation or expression.

2 Problem Specification

Nothing could better illustrate our approaches to optimizing the performance of a DSP through the use of a HLL than a simple, concrete example. The particular example chosen is a C function rmvmul(). rmvmul() performs a multiplication between a real matrix **A** and a real vector **x** yielding the real vector **y**, that is **y** = **Ax**. rmvmul() has the advantage of being short enough to be considered in detail in this paper and sufficiently complex to describe several optimization strategies. rmvmul() and its arguments are defined as follows:

SYNOPSIS

rmvmul(y, A, x, ncA, nrA)

DESCRIPTION

float	y[]	Pointer to real output vector.
float	A[]	Pointer to real input matrix.
float	x[]	Pointer to real input vector.
int	ncA	Number of columns in matrix A and rows in vector x.
int	nrA	Number of rows in matrix A and rows in vector y.

Returns −1 if **nca** ≤ 0 or **nra** ≤ 0.
Otherwise, returns 0.

3 C Versions of rmvmul()

3.1 Plain-Vanilla Version of rmvmul()

We will present several different versions of rmvmul(). Starting with our first version, the "plain-vanilla" version, we will follow a path of step-wise refinement in order to maximize the algorithm's performance. The plain-vanilla version is representative of the way one might write the routine without considering any optimization. The plain-vanilla version is shown in Figure 1. This version of rmvmul() executes in 24.36 μs on a TMS320C30 (@ 60ns/cycle).

The code in Figure 1 is divided into two portions. The first is the control portion of the code. This portion of the function checks for errors on the dimensions of the vector and matrices passed to it. If there is an error, rmvmul() returns −1. Otherwise, 0 is returned. This code is shown in Figure 2 and the resulting TMS320C30 assembly-language code is shown in Figure 3. This code is typical of the general-purpose code often found in a more sophisticated DSP-system. Notice that the compiler has produced extremely efficient code in this case.

The second major portion of the code is the computational portion of the code. This is the code contained within the **for** (i = 0; i < nrA; i++) {...} construct in Figure 1. Figure 4 shows the assembly-language code produced by the compiler for the statement sum += A[inca + j] * x[j]. It is in this eight lines of assembly-language code where most of the execution time is spent.

While the compiler has produced extremely efficient code from the C code we provided, it is possible to significantly improve the performance of the function by simply modifying our C code to take advantage of register variables.

3.2 Using Register Variables

The TMS320C30, due to its register-based CPU, supports the C **register** type very efficiently. **register** variables are stored in CPU registers and do not have to be stored in memory. The result is these variables are then accessed very efficiently whenever they are needed. The TMS320C30 C-compiler user has available two CPU registers for variables of type **register float**, two CPU registers for variables of type **register int**, and four registers for pointer register-variables. If we rewrite rmvmul() to take advantage of these registers, then the variable definitions for rmvmul() appear as shown in Figure 5. This register version of rmvmul() executes in 19.08 μs. This dramatic improvement is due to, in large part, the shortened time necessary to calculate sum += A[inca + j] * x[j]. The TMS320C30 assembly-language code for this calculation is shown in Figure 6.

In the use of register variables, we followed one simple rule: *use register variables for data that is accessed often.* In this case, the pointers to the vectors and matrix were made register variables since these addresses are used often in the inner loop. Since the variable sum and the loop indices i and j are accessed every iteration, they were declared as register variables.

4 Further Optimizations

Thus far we have focused only on the optimization of the C source-code by being smart about the way we write our algorithm. The next level of optimization involves optimization at the assembly-language level. The TMS320C30 C-compiler, assembler, and linker support several approaches for further optimization.

The C compiler supports the insertion of assembly-language code in the assembly-language output of the compiler with the asm() function. from the C source-code level. For example, asm("ADDF R0, R1") will result in the assembly-language code ADDF R0, R1 being inserted directly into the assembly-language output of the compiler. This technique is particularly useful for TMS320C30 system-level functions such as enabling and disabling interrupts and enabling and disabling the cache.

Since the output of the compiler is a TMS320C30 assembly-language source-file, this source file may be modified by the user. This approach may be a good starting-point for producing an optimized assembly-language module.

The third, and often best, way to optimize the C code, is to write time-critical functions in assembly language. By following the calling conventions of the C compiler, assembly-language functions can be written and called directly from the C source code. This approach is already common in the use of general-purpose microprocessors when speed is critical. This is the approach we will now follow.

4.1 Optimization With Assembly-Language Functions

As discussed previously, the heart of the rmvmul() function is

```
for (j = 0; j < ncA; j++)     /* Step across a row. */
     sum += A[inca + j] * x[j];
```

This is the dot product between the vector **x** and row i of matrix **A**. The TMS320C30 supports the dot-product operation extremely efficiently, and since this is where most of the time in the algorithm is spent, this is a very reasonable point to optimize with an assembly-language function. We will call this function dotpr() and define it as follows:

SYNOPSIS

```
float dotpr(x, y, n)
```

DESCRIPTION

float	x[]	Pointer to real input vector.
float	y[]	Pointer to real input vector.
int	n	Integer input length of vectors.

Returns the dot product of vectors x and y.

The assembly-language code for dotpr() is shown in Figure 7.

If we rewrite rmvmul() to take advantage of the assembly-language optimized dotpr() function, then we have the code shown in Figure 8. This version of rmvmul() executes in 14.22 μs.

Table 1 summarizes the results of the three different approaches we used in the step-wise optimization of rmvmul().

5 A 'Pure C' Benchmark

Sometimes, the DSP-system designer will find himself or herself with a large body of general-purpose code which has very little to do with the traditional multiply/accumulate intensive code one usually sees. For these cases it is important to understand how well the C compiler, in conjunction with the DSP, will perform.

The Dhrystone program is one measure of processor/compiler efficiency in executing a 'typical' program. The Dhrystone program has the following distribution of operations ([2]): assignments - 53%, control statements - 32%, and function calls - 15%. The Dhrystone contains no floating-point operations and does not contain much code that can be optimized by vector processors. Therefore, the Dhrystone can be used as one measure of how a processor/compiler combination would perform on the general-purpose code which is becoming more of a factor in DSP system-level applications.

Table 2 summarizes the number of Dhrystones/sec achieved by the TMS320C30 and some other familiar machines ([3]). A rate of 1 Dhrystone/sec would correspond to one pass through the Dhrystone program in one second. Therefore, the greater the number of Dhrystones/sec, the better the processor/compiler combination on this benchmark.

6 Conclusion

The TMS320C30 and its optimizing C-compiler achieve very high levels of performance on not only numerically intensive code but also general-purpose code. Given this, it is evident that HLLs are very useful tools for the DSP-system designer *when* the DSP and optimizing HLL compiler are developed with a strong emphasis on high performance and ease of use.

References

[1] R. Simar Jr., T. Leigh, P. Koeppen, J. Leach, J. Potts, and D. Blalock, "A 40 MFLOPS Digital Signal Processor: The First Supercomputer on a Chip," *Proc. 1987 IEEE Int. Conf. on Acous., Speech, and Signal Processing*, pp. 535-538, April 1987.

[2] R. Richardson, "Dhrystone Program," *BIX Bulletin Board*, January 1987.

[3] R. Richardson, "Dhrystone 1.1 Benchmark Summary," *BIX Bulletin Board*, January 1987.

Version	Time
Plain vanilla	24.36 μs
Register-variable optimization	19.08 μs
Assembly-language optimization	14.22 μs

Table 1: Summary of the results of the approaches to optimizing rmvmul().

Computer/Processor	Dhrystones/sec
Vax 11/780	1,640
Motorola 68020	3,257
Intel 80386	7,810
MIPS M/500	10,309
TMS320C30	10,987

Table 2: The TMS320C30 and C-compiler Dhrystone performance relative to some other computers and processors ([3]).

```
rmvmul(y, A, x, ncA, nrA)

float y[], A[], x[];
int   ncA, nrA;
{
    int   i,j;
    float sum;
    int   inca;

    /* Check for errors on dimensions. */
    if (ncA <= 0 || nrA <= 0)
        return(-1);

    for (i = 0; i < nrA; i++)     /* Step down a column. */
    {
        inca = i * ncA;
        sum  = 0.0;
        for (j = 0; j < ncA; j++) /* Step across a row. */
            sum += A[inca + j] * x[j];
        y[i] = sum;
    }
    return(0);
}
```

Figure 1: The plain-vanilla version of `rmvmul()`.

```
/* Check for errors on dimensions. */
if (ncA <= 0 || nrA <= 0)
    return(-1);
    {
    /* computations */
    }
return(0);
```

Figure 2: C source for control portion of `rmvmul()`.

```
_rmvmul: LDI     *-FP(5),R0      ;R0 = ncA
         BLE     LL3
         LDI     *-FP(6),R1      ;R1 = nrA
         BGT     L1
LL3:     LDI     -1,R0           ;return(-1)
         B       EPIO_2
L1:
    ; /* computations */
L3:      LDI     0,R0            ;return(0)
EPIO_2:  SUBI    4,SP
         POP     FP
         RETS
```

Figure 3: Resulting assembly-language code for the control portion of `rmvmul()`.

```
        LDI     *-FP(3),R1      ;*-FP(3) = A[]
        ADDI    *+FP(4),R1      ;*+FP(4) = inca
        ADDI    R0,R1,AR0       ;R0 = j
        ADDI    *-FP(4),R0      ;*-FP(4) = x[]
        LDI     R0,AR1
        MPYF    *AR1,*AR0,R0    ;R0 = A[inca + j] * x[j]
        ADDF    *+FP(3),R0      ;R0 += sum
        STF     R0,*+FP(3)      ;*+FP(3) = sum
```

Figure 4: Resulting assembly-language code for `sum += A[inca + j] * x[j]`

```
rmvmul(y, A, x, ncA, nrA)

register float y[], A[], x[];
int ncA, nrA;
{
    register int i,j;
    register float sum;
    int inca;
    {
    /* Error checking and computations */
    }
}
```

Figure 5: Variable definitions for `rmvmul()`when using register variables.

```
        LDI     *+FP(1),R0      ;*+FP(1) = inca
        ADDI    R0,AR5,R1       ;AR5 = A[]
        ADDI    R5,R1,AR0       ;R5 = j
        ADDI    R5,AR6,AR1      ;AR6 = x[]
        MPYF    *AR1,*AR0,R1    ;R1 = A[inca + j] * x[j]
        ADDF    R1,R6           ;R6 = sum
```

Figure 6: Resulting assembly-language source for register version of `sum += A[inca + j] * x[j]`

```
; Initialization
_dotpr: PUSH    FP              ; Save the old FP.
        LDI     SP,FP           ; Point to the top of the stack.
        LDI     *-FP(2), AR0    ; AR0 = x
        LDI     *-FP(3), AR1    ; AR1 = y
        LDI     *-FP(4), RC     ; RC = n
        SUBI    1, RC           ; RC = n-1
        LDF     0.0, R0         ; R0 = 0
        LDF     0.0, R2         ; R2 = 0
; Dot product
        RPTS    RC
        MPYF    *AR0++, *AR1++, R0   ; R0 = x[i++] * y[j++]
||      ADDF    R0,R2                ; R2 = sum' = sum + R0
        ADDF    R0,R2                ; Last product.
; Return
        LDF     R2, R0          ; Put the result in R0.
        POP     FP              ; Pop the old frame pointer.
        RETS
```

Figure 7: Assembly-language routine `dotpr()`.

```
rmvmul(y, A, x, ncA, nrA)

register float y[], A[], x[];
register int ncA;
int nrA;
{
    register int i;
    float dotpr();

    /* Check for errors on dimensions. */
    if (ncA <= 0 || nrA <= 0)
        return(-1);

    for (i = 0; i < nrA; i++)  /* Step down a column. */
        y[i] = dotpr(x, &A[i * ncA], ncA);

return(0);
}
```

Figure 8: C source of `rmvmul()`which uses the assembly language optimized routine `dotpr()`.