Optimizing Compilers and Embedded Software

By Rob Oshana

Introduction

Many of today's DSP applications are subject to real-time constraints. Many applications will eventually grow to a point where they are stressing the available CPU and memory resources. Understanding the workings of the DSP architecture, compiler, and the application algorithms can speed up applications, sometimes by an order of magnitude. This article will summarize some of the techniques that can improve the performance of your code in terms of cycle count, memory use, and power consumption.

What Is Optimization?

Optimization is a procedure that seeks to maximize or minimize one or more performance indices. These indices include;

- Throughput (execution speed)
- Memory usage
- I/O bandwidth
- Power dissipation

Since many DSP systems are real-time systems, at least one (and probably more) of these indices must be optimized. It is difficult (and usually impossible) to optimize all these performance indices at the same time. For example, making the application faster may require more memory and vice versa. The designer must weigh each of these indices and make the best tradeoff

Determining which index or set of indices are important to optimize depends on the goals of the application developer. For example, optimizing for performance means that the developer can use a slow or less expensive DSP to do the same amount of work. In some embedded systems, cost savings like this can have a significant impact on the

success of the product. The developer can alternatively choose to optimize the application to allow the addition of more functionality. This may be very important if the additional functionality improves the overall performance of the system, or if the developer can add more capability to the system such as an additional channel of a base station system. Optimizing for memory use can also lead to overall system cost reduction. Reducing the application size leads to a lower demand for memory which reduces overall system cost. And finally, optimizing for power means that the application can run longer on the same amount of power. This is important for battery powered applications. This type of optimization also reduces the overall system cost with respect to power supply requirements and other cooling functionality required.

The tricky part to optimizing DSP applications is to understand the tradeoff between the various performance indices. For example, optimizing an application for speed often means a corresponding decrease in power consumption but an increase in memory usage. Optimizing for memory may also result in a decrease in power consumption due to fewer memory accesses but a offsetting decrease in code performance. The various tradeoffs and system goals must be understood and considered before attempting any form of application optimization.

Make The Common Case Fast

The fundamental rule in computer design as well as programming real-time DSP-based systems is "make the common case fast, and favor the frequent case". This is really just Amdahl's Law that says the performance improvement to be gained using some faster mode of execution is limited by how often you use that faster mode of execution. So don't spend time trying to optimize a piece of code that will hardly ever run. You won't get much out of it, no matter how innovative you are. Instead, if you can eliminate just one cycle from a loop that executes thousands of times, you will see a bigger impact on the bottom line.

1. Make the Common Case Fast – DSP Architectures

DSP architectures are designed to make the common case fast. Considering many DSP applications are composed from a standard set of DSP building blocks such as filters, Fourier Transforms, and convolutions. These algorithms all share a common

characteristic; they perform multiplies and adds over and over again (Figure 1). This is generally referred to as the Sum of Products (SOP). DSP chip designers have developed hardware architectures that allow the efficient execution of algorithms with SOPs. This is done using specialized instructions such as single cycle multiple and accumulate (MAC), architectures that all multiple memory accessed in a single cycle (Harvard achitectures, Figure 2) and special hardware that handles loop counting with very little overhead.

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-2\pi i x u / N}, \qquad (13)$$

Discrete Fourier Transform

$$y(n) = \sum_{k=0}^{M-1} b_k \cdot x(n-k) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k)$$

Filter algorithm

Figure 1 – DSP algorithms are composed of iterations of multiplies and adds

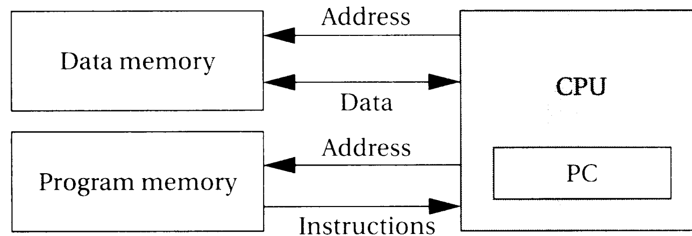

Figure 2 – Harvard Architecture. The separation of program and data provides increased performance for DSP applications (Wolf, page 59)

2. Make the Common Case Fast – DSP Algorithms

DSP algorithms can be made to run faster using techniques of algorithmic transformation. For example, a common algorithm used in DSP applications is the Fourier Transform. The Fourier Transform is a mathematical method of breaking a

signal in the time domain into all of its individual frequency components[1]. The process of examining a time signal broken down into its individual frequency components is also called spectral analysis or harmonic analysis.

There are different ways to characterize a Fourier transforms;

- The Fourier Transform (FT) is a mathematical formula using integrals

$$F(u) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x u} \, dx. \qquad (7)$$

- The Discrete Fourier Transform (DFT) is a discrete numerical equivalent using sums instead of integrals which maps well to a digital processor like a DSP

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-2\pi i x u / N}, \qquad (13)$$

- The Fast Fourier Transform (FFT) is just a computationally fast way to calculate the DFT which reduces many of the redundant computations of the DFT.

How these are implemented on a DSP has a significant impact on overall performance of the algorithm. The FFT, for example, is a fast version of the DFT. The FFT makes use of periodicities in the sines that are multiplied to perform the transform. This significantly reduces the amount of calculations required. A DFT implementation requires N^2 operations to calculate a N point transform. For the same N point data set, using a FFT algorithm requires N * log2(N) operations. The FFT is therefore faster than the DFT by a factor of N/log2(n). The speedup for a FFT is more significant as N increases (Figure 3).

---

[1] Brigham, E. Oren, 1988, The Fast Fourier Transform and Its Applications, Englewood Cliffs, NJ: Prentice-Hall, Inc., 448 pp.

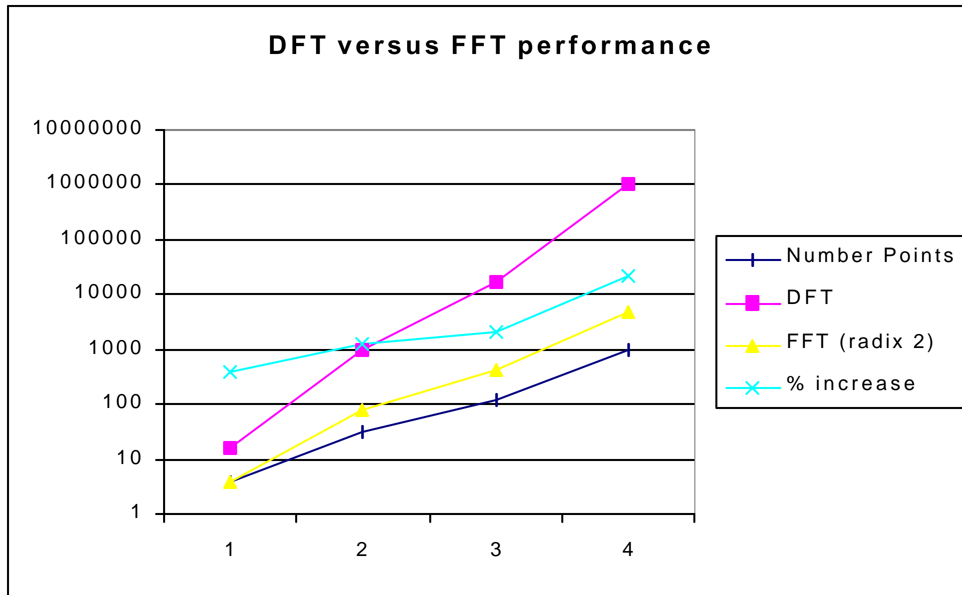

Figure 3. FFT vs DFT for various sizes of transforms (logarithmic scale)

Recognizing the significant impact that efficiently implemented algorithms have on overall system performance, DSP vendors and other providers have developed libraries of efficient DSP algorithms optimized for specific DSP architectures. Depending on the type of algorithm, these can downloaded from web sites (be careful of obtaining free software like this – the code may be buggy as there is no guarantee of quality) or bought from DSP solution providers.

3. Make the Common Case Fast – DSP Compilers

Just a few years ago, it was an unwritten rule that writing programs in assembly would usually result in better performance than writing in higher level languages like C or C++. The early “optimizing” compilers solved the problem of "optimization" at too general and simplistic a level. The results were not as good as a good assembly language programmer. Compilers have gotten much better and today there are very specific high performance optimizations performed that compete well with even the best assembly language programmers.

A general optimization strategy is to write DSP application code that can be *pipelined* efficiently by the compiler.. *Software* pipelining is an optimization strategy to schedule loops and functional units efficiently. In the case of the C6200 family of DSP,

there are eight functional units that can be used at the same time (Figure 4). Its up to the compiler to figure out how to schedule instructions on all of these units for each clock cycle. Sometimes is a matter of a subtle change in the way the C code is structured that makes all the difference. In software pipelining, multiple iterations of a loop are scheduled to execute in parallel. The loop is reorganized in a way that each iteration in the pipelined code is made from instruction sequences selected from different iterations in the original loop.

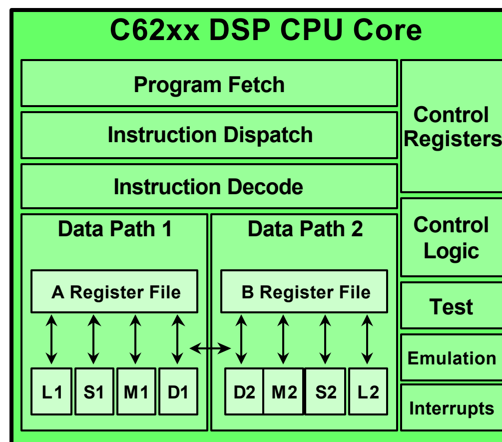

Figure 4. DSP Architectures may have orthogonal execution units and data paths used to execute DSP algorithms more efficiently

Figure 5 shows a sample piece of C code and the corresponding assembly language output. In this example, the pipelined code is not as good as it could be. You can spot inefficient code by looking for how many NOPs are in the piped loop kernel of the code. In this case the piped loop kernel has a total of 5 NOP cycles, 2 in line 16, and 3 in line 20. This loop takes a total of 10 cycles to execute. The NOPs are the first indication that a more efficient loop may be possible.

```
   1 void example1(float *out, float *input1, float
*input2)
   2 {
   3   int i;
   4
   5   for(i = 0; i < 100; i++)
   6    {
```

```
   7       out[i] = input1[i] * input2[i];
   8    }
   9 }

   1 _example1:
   2 ;** -----------------------------------------------------
-------*
   3           MVK        .S2       0x64,B0
   4
   5           MVC        .S2       CSR,B6
   6 ||        MV         .L1X      B4,A3
   7 ||        MV         .L2X      A6,B5

   8           AND        .L1X      -2,B6,A0
   9           MVC     .S2X      A0,CSR
   10    ;** ---------------------------------------------
------------*
   11    L11:        ; PIPED LOOP PROLOG
   12    ;** ---------------------------------------------
------------*
   13    L12:        ; PIPED LOOP KERNEL

   14              LDW        .D2       *B5++,B4      ;
   15    ||        LDW        .D1       *A3++,A0     ;

   16              NOP                  2
   17    [ B0]     SUB        .L2       B0,1,B0       ;
   18   [ B0]     B          .S2       L12            ;
   19              MPYSP      .M1X      B4,A0,A0     ;
   20              NOP                  3
   21              STW        .D1       A0,*A4++    ;
   22    ;** ---------------------------------------------
-----------*
   23              MVC        .S2       B6,CSR
   24              B          .S2       B3
   25              NOP                  5
   26              ; BRANCH OCCURS
```

Figure 5. C example and the corresponding pipelined assembly language output

In this example, the compiler did not optimize the loop efficiently because of the possible dependencies on the two input arrays. The programmer can explicity indicate in the code that these two arrays are not dependent and the compiler will apply more aggressive software pipelining optimizations to this code. By declaring the *input1* and

input2 arrays as "const" (see the code fragment below) allows the compiler to enable software pipelining which reduces the number of overall cycles to complete the operation. This C code is shown in Figure 6 with the corresponding assembly language.

```
1  void example2(float *out, const float *input1, const float *input2)
2  {
3    int i;
4
5    for(i = 0; i < 100; i++)
6     {
7        out[i] = input1[i] * input2[i];
8     }
9  }
```

```
1  _example2:
2  ;** ---------------------------------------------------------------*
3            MVK          .S2           0x64,B0
4
5            MVC          .S2           CSR,B6
6  ||        MV           .L1X          B4,A3
7  ||        MV           .L2X          A6,B5
8
9            AND          .L1X         -2,B6,A0
10
11           MVC          .S2X         A0,CSR
12 ||        SUB          .L2          B0,4,B0
13
14 ;** ---------------------------------------------------------------*
15 L8:       ; PIPED LOOP PROLOG
16
17           LDW          .D2          *B5++,B4      ;
18 ||        LDW          .D1          *A3++,A0      ;
19
20           NOP                       1
21
22           LDW          .D2          *B5++,B4      ;@
23 ||        LDW          .D1          *A3++,A0      ;@
24
25    [ B0]  SUB          .L2          B0,1,B0       ;
26
27    [ B0]  B            .S2          L9            ;
28 ||        LDW          .D2          *B5++,B4      ;@@
29 ||        LDW          .D1          *A3++,A0      ;@@
30
31           MPYSP        .M1X         B4,A0,A5       ;
32 || [ B0]  SUB          .L2          B0,1,B0        ;@
33
34    [ B0]  B            .S2          L9             ;@
35 ||        LDW          .D2          *B5++,B4       ;@@@
36 ||        LDW          .D1          *A3++,A0       ;@@@
37
38           MPYSP        .M1X         B4,A0,A5       ;@
39 || [ B0]  SUB          .L2          B0,1,B0        ;@@
40
41 ;** ---------------------------------------------------------------*
42 L9:       ; PIPED LOOP KERNEL
43
44    [ B0]  B            .S2          L9            ;@@
```

```
45 ||           LDW           .D2          *B5++,B4      ;@@@@
46 ||           LDW           .D1          *A3++,A0      ;@@@@
47
48              STW           .D1          A5,*A4++      ;
49 ||           MPYSP         .M1X         B4,A0,A5      ;@@
50 || [ B0]     SUB           .L2          B0,1,B0       ;@@@
51
52 ;** -------------------------------------------------------------*
53 L10:           ; PIPED LOOP EPILOG
54              NOP                        1
55
56              STW           .D1          A5,*A4++      ;@
57 ||           MPYSP         .M1X         B4,A0,A5      ;@@@
58
59              NOP                        1
60
61              STW           .D1          A5,*A4++      ;@@
62 ||           MPYSP         .M1X         B4,A0,A5      ;@@@@

64              NOP                        1
65              STW           .D1          A5,*A4++      ;@@@
66              NOP                        1
67              STW           .D1          A5,*A4++      ;@@@@
68 ;** -------------------------------------------------------------*
69               MVC          .S2          B6,CSR
70               B            .S2          B3
71               NOP                       5
72              ; BRANCH OCCURS
```

Figure 6. Corresponding pipelined assembly language output (the loop is now only 2 cycles long which dramatically improves performance for large loops)

The code size for a pipelined function becomes larger, which is obvious by looking at the output assembly code shown in the example. This is one of the tradeoffs (code size versus speed) that the embedded programmer must make.

Summary

Embedded real-time applications are an exercise in optimization. There are three main optimization strategies that the embedded DSP developer needs to consider;

- DSP architecture optimization; DSPs are optimized microprocessors that perform signal processing functions very efficiently by providing hardware support for common DSP functions,
- DSP algorithm optimization; choosing the right implementation technique for standard and often used DSP algorithms can have a significant impact on system performance,

- DSP complier optimization; DSP compilers are tools that help the embedded programmer exploit the DSP architecture by mapping code onto the resources in such as way as to utilize as much of the processing resources as possible, gaining the highest level of architecture entitlement as possible.