

Making The Blackfin Perform

by Robin Getz, Engineering Manager, Open Source Projects, Analog Devices

Optimizing source code to run on a specific platform can be challenging. While code-creation tools are improving, they haven't kept up with the rapidly increasing functionality and complexity of processors. The more complex the hardware architecture, the harder it is to program in assembly language.

This creates the need for abstraction via a robust C compiler or operating system. But since not all compilers handle source code in the same manner, you may need to rewrite your C source many times to achieve an efficient output.

While trial and test are valuable for optimizing source code, other methods exist. First, the compiler can use built-in functions. Second, C-callable specific libraries can be implemented. The combination of built-in functions and core-specific libraries will improve the code performance without getting involved in the complexity of the architecture.

Take Analog Devices' Blackfin processor, which is supported by open-source tools such as a gcc compiler and uClinux kernel. Recent efforts to incorporate built-in functions in the compiler and port several signal-processing-specific libraries have greatly improved its code efficiency.

Measuring Performance

Like many modern processor architectures, Blackfin has a complex [memory](#) architecture that uses L1 (cache and no-cache), L2, and L3. How the memory is configured and where the code is running from will impact execution speed more than any other parameter in the system.

In a typical system, cache will be turned on, and the code in question will be run a few times. As a result, it can be pre-loaded into cache, which avoids having to measure the time of loading things into cache. The basic method is:

```
foo () {
    code to be measured
}

main () {
    foo()          /* run once - load into cache */
    start=clock(); /* capture clock                */
    foo()          /* take measurement                */
    stop=clock();  /* capture clock                */
    interval=stop-start; /* find the time                */
    print interval; /* print the results            */
}
```

This is done with:

file: [uClinux-dist/testsuites/performance/dot.c:cycles\(\)](http://uClinux-dist/testsuites/performance/dot.c:cycles())

```

int cycles()
{
    int ret;

    __asm__ __volatile__ ("%0 = CYCLES;\n\t"
        : "=d"(ret));

    return ret;
}

```

and is used like:

file: [uClinux-dist/testsuites/performance/dot.c:test1\(\)](http://uClinux-dist/testsuites/performance/dot.c:test1())

```

int test1()
{
    short x[N];
    short y[N];
    int i, k, ret;
    unsigned int before, after;
    unsigned int time[M];

    for (i = 0; i < N; i++) {
        x[i] = 1;
        y[i] = 1;
    }

    for (k = 0; k < M; k++) {
        before = cycles();
        ret = dot_generic(x, y, N);
        after = cycles();
        time[k] = after - before;
    }

    printf("Test 1: Vanilla C\n");
    printf("  ret = %d: run time:\n ", ret);
    for (k = 0; k < M; k++)
        printf("%u ", time[k]);
    printf("\n");
    return ret;
}

```

Example: Dot Product

A **dot product** is a very common mathematical operation that takes two vectors and returns a scalar quantity in many signal-processing algorithms. For example:

$$\text{If } \mathbf{x} = [x_0, x_1, x_2, x_3, x_4, \dots, x_n] \text{ and } \mathbf{y} = [y_0, y_1, y_2, y_3, y_4, \dots, y_n]$$

then the dot product would be:

$$\mathbf{x} \odot \mathbf{y} = x_0 y_0 + x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4 + \dots + x_n y_n = \sum_{i=0}^n x_i y_i$$

This is represented in generic, portable C code as:

file: [uClinux-dist/testsuites/performance/dot.c:dot_generic\(\)](http://uClinux-dist/testsuites/performance/dot.c:dot_generic())

```
int dot_generic(short *x, short *y, int len)
{
    int i, dot;

    dot = 0;
    for (i = 0; i < len; i++)
        dot += x[i] * y[i];

    return dot;
}
```

This performance is close to (in each case, a low number of cycles is better):

```
root:/var/tmp> ./cycles
Theoretical best case is N/2 = 50 cycles
Test 1: Vanilla C
  ret = 100: run time:
  3838 3507 3373 3408 3373 3373 3373 3373 3373 3373
```

Notice that the first two times that the application is run, the number of cycles takes longer than normal, since the function and data aren't in an instruction or data cache.

Since many signal-processing applications use a native multiply-accumulate (MAC), simply using the assembly instructions to perform the dot product may not approach the theoretical limit of the processor (which in the Blackfin case is a half-MAC per cycle).

Changing to assembly provides:

file: [uClinux-dist/testsuites/performance/dot.c:dot_asm\(\)](http://uClinux-dist/testsuites/performance/dot.c:dot_asm())

```
int dot_asm(short *x, short *y, int len)
{
    int dot;

    /*
     * Isn't it cool how you can mix C and assembler?  And gcc just
     * takes care of all the fiddly little details.
     * Very nice
     */

    __asm__ ("I0 = %3;\n\t"
            "I1 = %4;\n\t"
            "A1 = A0 = 0;\n\t"
            "R0 = [I0++] || R1 = [I1++];\n\t"
            "LOOP dot%= LC0 = %5 >> 1;\n\t"
            "LOOP_BEGIN dot%=\n\t"
            "A1 += R0.H * R1.H, A0 += R0.L * R1.L || R0 = [I0++] ||
R1 = [I1++];\n\t"
            "LOOP_END dot%=\n\t"
            "R0 = (A0 += A1);\n\t"
            "%0 = R0 >> 1;\n\t" /* correct for left shift during
multiply */
            : "=&d" (dot), "=&d" (before), "=&d" (after)
            : "a" (x), "a" (y), "a" (len))
}
```

```

        : "I0", "I1", "A1", "A0", "R0", "R1");
    return dot;
}

```

This provides an output of:

```

Test 2: data in external memory, outboard cycles function
ret = 100: run time:
442 240 239 218 218 218 218 218 218 218

```

Consequently, performance improves by more than 15 times! However, this code is now completely optimized for the Blackfin. It can't be run on any other architecture. And, it's still four times over the theoretical performance. Because this function uses both the instruction and data cache, some optimizations still can be done.

The next step is to add the cycle's measurement into the same function, which actually does the dot product:

file: [uClinux-dist/testsuites/performance/dot.c:dot_asm_cycles\(\)](http://uClinux-dist/testsuites/performance/dot.c:dot_asm_cycles())

```

int dot_asm_cycles(short *x, short *y, int len)
{
    int dot;

    __asm__ ("I0 = %3;\n\t"
            "I1 = %4;\n\t"
            "A1 = A0 = 0;\n\t"
            "R0 = [I0++] || R1 = [I1++];\n\t"
            "R2 = CYCLES;\n\t"
            "%1 = R2;\n\t"
            "LOOP dot%= LC0 = %5 >> 1;\n\t"
            "LOOP_BEGIN dot%=;\n\t"
            "A1 += R0.H * R1.H, A0 += R0.L * R1.L || R0 = [I0++] ||
R1 = [I1++];\n\t"
            "LOOP_END dot%=;\n\t"
            "R2 = CYCLES;\n\t"
            "%2 = R2;\n\t"
            "R0 = (A0 += A1);\n\t"
            "%0 = R0 >> 1;\n\t" /* correct for left shift during
multiply */
            : "=&d" (dot), "=&d" (before), "=&d" (after)
            : "a" (x), "a" (y), "a" (len)
            : "I0", "I1", "A1", "A0", "R0", "R1", "R2");

    return dot;
}

```

This function doesn't have the overhead of calling a function to measure the cycles of the MAC, as does the dot product. Its output is:

```

Test 3: data in external memory, inboard cycles
ret = 100: run time:
242 103 103 103 103 103 103 103 103 103

```

While this still approaches the theoretical maximum, it's still twice what it should be due to the processor's memory structure. The main part of the hardware loop involves a parallel instruction `A1 += R0.H * R1.H, A0 += R0.L * R1.L || R0 = [I0++] || R1 = [I1++]`; where the two loads are from `I0` and `I1`. If both point to the same

memory bank, then the access will be stalled one instruction, as the two load data buses come from L1 Bank A and L1 Bank B.

The code shouldn't be written this way. Yet at the time of publication, there weren't 11 malloc functions for obtaining blocks of internal memory.

This can be shown with:

file: [uClinux-dist/testsuites/performance/dot.c:test4\(\)](https://uclinux-dist/testsuites/performance/dot.c:test4())

```
int test4()
{
    /* I know, I know - this is very naughty :-) */
    short *x = (short *)0xff904000 - N * sizeof(short); /* Top of
Data B SRAM */
    short *y = (short *)0xff804000 - N * sizeof(short); /* Top of
Data A SRAM */

    int i, k, ret;
    unsigned int time[M];

    for (i = 0; i < N; i++) {
        x[i] = 1;
        y[i] = 1;
    }

    for (k = 0; k < M; k++) {
        ret = dot_asm_cycles(x, y, N);
        time[k] = after - before;
    }

    printf("Test 4: data in internal memory, inboard cycles\n");
    printf("  ret = %d: run time:\n  ", ret);
    for (k = 0; k < M; k++)
        printf("%u ", time[k]);
    printf("\n");
    return ret;
}
```

which ensures that the two vectors are in different data banks. It provides an output close to the theoretical maximum.

```
Test 4: data in internal memory, inboard cycles
  ret = 100: run time:
  214 53 53 53 53 53 53 53 53 53
```

A 100-point dot product should take 50 clock cycles on a Blackfin. The code runs four test cases and manages to reduce the execution time from 3838 cycles to 53 cycles through various tricks.

Each test runs 10 times. In several of the tests, you can see the number of cycles reducing as the instruction and data cache gets loaded over successive runs.

This example shows that even if you hand-write functions in assembly, it's easy to slow your application by a factor of two to four by not understanding the chip's implementation of the memory structure, or simply by calling a function. With a little

optimization and some hand-coded assembler, it's possible to get full performance from the chip.

Typically, writing hand-optimizing assembler sounds like a terrible idea; most people don't enjoy it much. Yet it can be worth the effort for the few "inner loop" routines, which are run many times and fit entirely in cache. Many pre-written signal-processing algorithms are available for designers who don't want to write their own assembler routines.

Example: FFT

Another good example of a commonly used function is the [fast Fourier transform](#) (FFT). This efficient algorithm computes the discrete Fourier transform (DFT) and its inverse. FFTs are very important to a wide variety of applications, from signal processing to solving partial differential equations to algorithms for quickly multiplying large integers.

If $x = x_0, x_1, \dots, x_{N-1}$ of complex numbers, the DFT is:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \quad k=0, \dots, N-1$$

\\?

Evaluating these sums normally would take $O(N^2)$ arithmetical operations, or the algorithm has order of n^2 time complexity. There are numerous methods and algorithms for calculating an FFT. [Cooley Tukey](#) is one of the most common.

file: [uClinux-dist/user/ndso/src_ndso/int_fft.c:fix_fft\(\)](#)

```
fix_fft (fixed fr[], fixed fi[], int m, int inverse)
{
    int mr, nn, i, j, l, k, istep, n, scale, shift;
    fixed qr, qi, tr, ti, wr, wi, t;

    n = 1 << m;

    if (n > N_WAVE)
        return -1;

    mr = 0;
    nn = n - 1;
    scale = 0;

    /* decimation in time - re-order data */
    for (m = 1; m <= nn; ++m)
    {
        l = n;
```

```

do
  {
    l >>= 1;
  }
while (mr + l > nn);
mr = (mr & (l - 1)) + 1;

if (mr <= m)
  continue;
tr = fr[m];
fr[m] = fr[mr];
fr[mr] = tr;
ti = fi[m];
fi[m] = fi[mr];
fi[mr] = ti;
}

l = 1;
k = LOG2_N_WAVE - 1;
while (l < n)
  {
    if (inverse)
      {
        /* variable scaling, depending upon data */
        shift = 0;
        for (i = 0; i < n; ++i)
          {
            j = fr[i];
            if (j < 0)
              j = -j;
            m = fi[i];
            if (m < 0)
              m = -m;
            if (j > 16383 || m > 16383)
              {
                shift = 1;
                break;
              }
          }
        if (shift)
          ++scale;
      }
    else
      {
        /* fixed scaling, for proper normalization -
           there will be log2(n) passes, so this
           results in an overall factor of 1/n,
           distributed to maximize arithmetic accuracy. */
        shift = 1;
      }
    /* it may not be obvious, but the shift will be performed
       on each data point exactly once, during this pass. */
    istep = 1 << 1;
    for (m = 0; m < l; ++m)
      {
        j = m << k;
        /* 0 <= j < N_WAVE/2 */

```

```

wr = Sinewave[j + N_WAVE / 4];
wi = -Sinewave[j];
if (inverse)
    wi = -wi;
if (shift)
    {
        wr >>= 1;
        wi >>= 1;
    }
for (i = m; i < n; i += istep)
    {
        j = i + 1;
        tr = fix_mpy (wr, fr[j]) - fix_mpy (wi, fi[j]);
        ti = fix_mpy (wr, fi[j]) + fix_mpy (wi, fr[j]);
        qr = fr[i];
        qi = fi[i];
        if (shift)
            {
                qr >>= 1;
                qi >>= 1;
            }
        fr[j] = qr - tr;
        fi[j] = qi - ti;
        fr[i] = qr + tr;
        fi[i] = qi + ti;
    }
}
--k;
l = istep;
}

return scale;
}

```

A alternative solution is to use the built in FFT function:

```

void cfft_fr16(in[], t[], out[], w[], wst, n,
block_exponent, scale_method)
    const complex_fract16 in[]; /* input sequence */
    complex_fract16 t[]; /* temporary working buffer */
    complex_fract16 out[]; /* output sequence */
    const complex_fract16 w[] /* twiddle sequence */
    int wst; /* twiddle factor stride */
    int n; /* number of FFT points */
    int block_exponent; /* block exponent of output data */
    int scale_method; /* scaling method desired:
                        0-none, 1-static, 2-dynamic */

```

The two different functions are functionally equivalent. But from a performance standpoint, it's more than an nx difference.