About this Module

This module introduces concepts, tools and approaches to optimising Blackfin C applications. It highlights the problems and opportunities that can lead to potentially significant performance improvements.

The course will look at the available tools, such as the Statistical profiler and then review techniques for handling Control code, DSP kernel code and memory, finishing with an example. The material covers a wide range from automatic optimisation to detailed rewriting of C expressions.

Module Outline

- **Concepts and Tools**
  - Concerns of C : Advantages & problems
  - Useful tools
  - Pipelines and stalls
- **Tuning DSP Kernels**
  - Tuning techniques
  - Advanced loop optimisation
- **Tuning Control code**
  - Introduction to optimising control code
  - Advanced control code optimisation
- **Blackfin Memory Performance**
  - Memory timing
  - Optimising for speed or space
- **Examples**
  - Data structure
  - Whole application example

Concepts and Tools

Why use C?

- **Advantages:**
  - C is much cheaper to develop. (encourages experimentation)
  - C is much cheaper to maintain.
  - C is comparatively portable.

- **Disadvantages:**
  - ANSI C is not designed for DSP.
  - DSP processor designs usually expect assembly in key areas.
  - DSP applications continue to evolve. (faster than ANSI Standard C)

How to go about increasing C performance

(1) Work at high level first
most effective -- maintains portability

- improve algorithm
- make sure it's suited to hardware architecture
- check on generality and aliasing problems

(2) Look at machine capabilities

- may have specialised instructions (library/portable)
- check handling of DSP-specific demands

(3) Non-portable changes last

- in C?
- in assembly language?
- always make sure simple C models exist for verification

Usually the process of performance tuning is a *specialisation* of the program for particular hardware. The program may become larger, more complex and less portable.

Uniform C computational model, BUT....

- **Missing operations provided by software emulation (floating point!)**
- **Assumes Large flat memory model**
- **C is more machine-dependent than you might think**
  - **for example: is a "short" 16 or 32 bits?**
- **Can be a poor match for DSP – accumulators? SIMD? Fractions?**
- **Not really a mathematical focus. Systems programming language**
- **Machine's Characteristics will determine your success.**
  - **What is the computational bandwidth and throughput?**
  - **Macs? Bus capacity? Memory access times?**

C programs can be ported with little difficulty.

But if you want high efficiency, you can't ignore the underlying hardware.

Two kinds of "Optimisation". Use the Optimiser *and* Optimise the C!

(1) Automatic compiler optimisation.

- Up to 20 times faster that non-optimised code in DSP kernels
- Sliding scale from control code to DSP inner loop
- Non-optimized code is only for debugging the algorithm

But Note: Optimising Compiler has Limited Scope

- will not make global changes
- will not substitute a different algorithm
- will not significantly rearrange data or use different types
- **Correctness as defined in the language is the priority**

(2) Elaborate the "out of the box" portable C, to "optimised C".

- Annotations: #pragmas, built-in functions, Memory qualifiers – const, restrict, volatile, bank
- Amendments: Targeted rewrites of the C statements, asm() statements

Un-Optimized Code for Blackfin

Unoptimised assembly:

```
  [FP+ -8] = R7; ._P1L1:
  R3=[FP+ -8];
  R2 =  150 (X);
  CC = R3 < R2;
  IF !CC JUMP  ._P1L3 ;
  R3 <<=  1;
  P2 = R3 ;
  P0=[FP+ 8];
  P0 = P0 + P2;
  R1=W[P0+ 0] (X);
  R0=[FP+ -8];
  R0 <<=  1;
  P1 = R0 ;
  P2=[FP+ 12];
  P2 = P2 + P1;
  R7=W[P2+ 0] (X);
  R7 *= R1 ;
  R1=[FP+ -4];
  R0 = R1 + R7;
 [FP+ -4] = R0;
  R3=[FP+ -8];
  R3 <<=  1;
  P0 = R3 ;
  P1=[FP+ 12];
  P1 = P1 + P0;
  R1=W[P1+ 0] (X);
  R7=[FP+ -8];
  R7 <<=  1;
  P2 = R7 ;
  P1=[FP+ 12];
  P1 = P1 + P2;
  R3=W[P1+ 0] (X);
  R3 *= R1 ;
  R1=[FP+ 16];
  R7 = R1 + R3;
  [FP+ 16] = R7;
  R3=[FP+ -8];
  R3 +=  1;
  [FP+ -8] = R3;
  JUMP ._P1L1;
```

Loop control
increment, test & exit

Load a[I]

Load b[I]

dotp += b[I]* a[I]

Load b[I]

Load b[I]

sqr += b[I] * b[I]

Increment I

Repeat Loop

The source code:

```
for (i = 0; i < 150; i++) {
   dotp += b[i] * a[i];
   sqr  += b[i] * b[i];
 }
```

The Optimised code (-O)

- easier to understand!

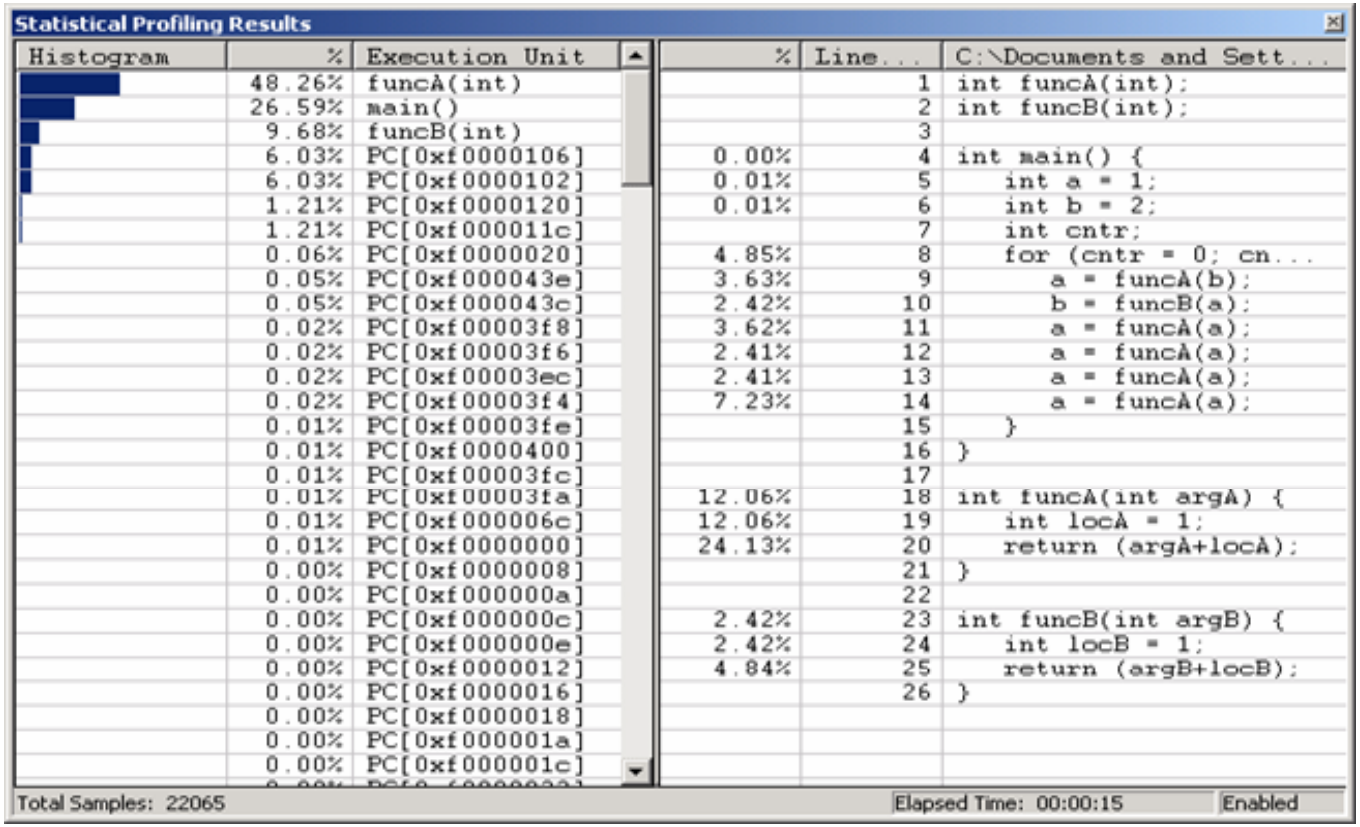
```
LSETUP (._P1L2 , ._P1L3-8) LC0=P1;

._P1L2:
  A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS)
          || R0.L = W[I1++]
          || R0.H = W[I0++];
._P1L3:
```

Direct your effort: Use the Statistical Profiler.

- **Statistical profiling samples the program counter of the running application and builds up a picture of where it spends its time**
- **Completely non-intrusive – no tracing code is added**
- **Completely accurate – shows all effects, including stalls, from all causes**
- **Do not assume that you know where an application spends its time**
- **Measure it:** **Intuition is notoriously bad here.**
- **80 – 20 rule: 80% of anything is usually non-critical.**

VDSP++ Statistical Profiler

- **Benchmark by function, then drill down**
- **Left pane shows line by line activity**



- **Linear Profiler is also available in the simulator**

A tip: Use Mixed Mode.

Statistical results at the instruction level.

Costly instructions are easy to spot.

1.24%	308	cosine = (double)(2.0 * cos((2 * v_1 + 1) * cosMultB
0.04%		[FFA00966] P0 = [FP + -72] ;
0.17%		[FFA00968] R0 = [P0 + 0x0] ;
0.04%		[FFA0096A] R0 <<= 0x1 ;
0.04%		[FFA0096C] R0 += 1 ;
0.21%		[FFA0096E] CALL ___int32_to_float32 ;
0.04%		[FFA00972] R1 = R0 ;
0.04%		[FFA00974] R0 = R6 ;
0.21%		[FFA00976] CALL ___float32_mul ;
0.21%		[FFA0098A] CALL __Sin ;
0.04%		[FFA0098E] R1 = R0 ;
0.21%		[FFA00990] CALL ___float32_add ;
	309	}
	310	else

<- Pipeline stalls

<- Transfer of control

Could mean pipeline or multi-cycle instruction or memory cost.

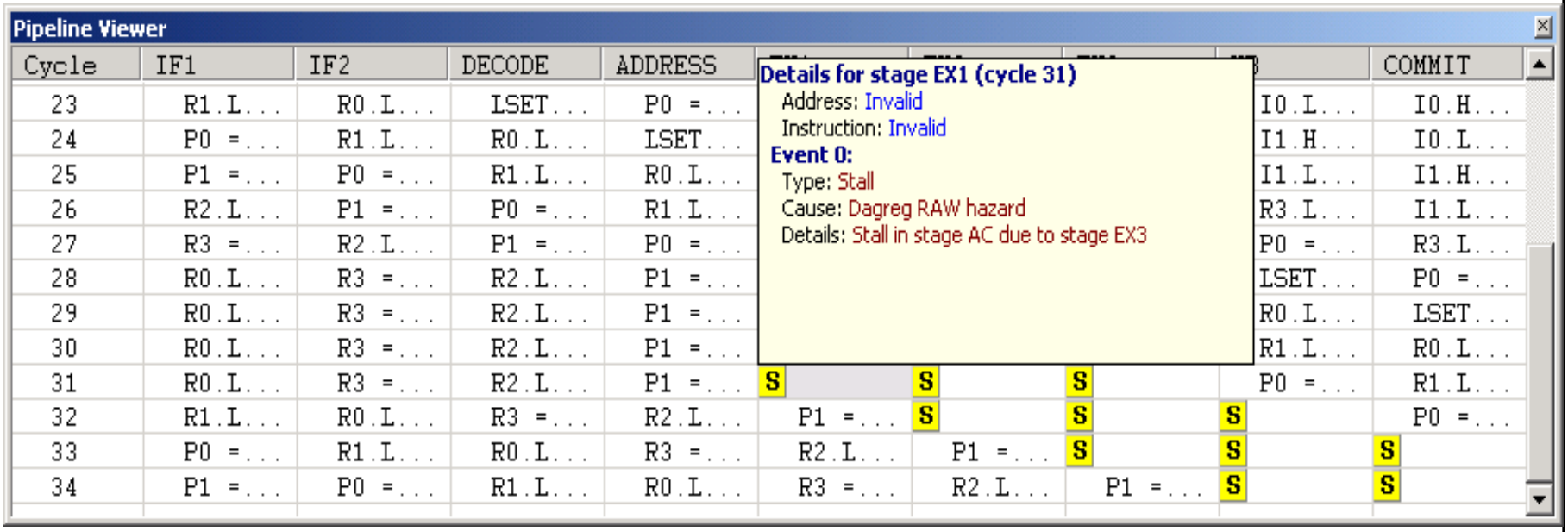
Due to pipeline, costs may be offset from adjacent instruction.

Bear in mind the Length of the Pipeline.

- **Take care on conditionally branching code. (0, 4 or 8 stall cycles)**
  - **sometimes branches can be avoided by using other techniques**
- **Take care with Table lookup and structure depth.**
  - **p->q->z is inefficient to access. (3 stalls per indirection)**
  - **(And also hard on pointer analysis. What data does this reference?)**
- **Is there a latency associated with computations? (results not ready on next cycle)**
- **C Compiler will do its best to schedule other code into stall cycles, but inherent hardware properties will always influence the outcome.**
- **EE197 note details Blackfin stalls.**

VDSP++ Pipeline Viewer (SIMULATION ONLY)

- **Accessed through View->Debug Windows->Pipeline Viewer.**



Press <Ctrl> and hover over stall.

Step through interesting code sequences and look out for the slash of yellow that means stalls.

Tuning Digital Signal Processing Kernels

An efficient floating Point Emulation

Measurement in cycles

VDSP++ 4.0

	BF533 fast-fp	BF533 ieee-fp	ratio
Multiply	93	241	0.4
Add	127	264	0.5
Subtract	161	329	0.5
Divide	256	945	0.3
Sine	2271	4665	0.5
Cos	2170	4419	0.5
Square Root	318	322	1.0
atan	1895	5712	0.3
atan2	2666	8058	0.3
pow	8158	17037	0.5

Smaller is better in cycles.

Note: The BF Square root uses a new algorithm.

The –fast-fp BF implementation relaxes strict IEEE checking of NAN values.

Wide support for Fractional processing

- **The Blackfin instruction set includes a number of operations which support fractional (or fract) data. The instructions include:**
  - **saturating MAC/ALU/SHIFT instructions**
  - **MAC shift correction for fractional inputs**
- **The compiler and libraries provide support for fractional types:**
  - **fractional builtins**
  - **fract types fract16 and fract32**
  - **ETSI**
  - **C++ fract class**
- **Fractional arithmetic is a hundred times faster than floating!**

Portable C expressing fractions

```
void vec_mpy1(short y[], const short x[], short scaler)
{
   int        i;
   for (i =  0; i < 150; i++)
      y[i] +=  ((scaler * x[i]) >> 15);
}
```

It is portable and the compiler is going to understand this, and generate optimal code for 16 bit fractions. (it will assume saturation)

But…..

- **C does not specify what happens on signed overflow. Some programs expect saturation, some expect to wrap to zero.**
- **Unsigned variables are stricter than signed.**
- **You do not have an extended precision accumulator type, so you cannot assume 40 bit precision. Which implies controlling range throughout the loop.**
- **More complex C expressions may stress the compilers ability to recognise what you are getting at. So the intrinsics offer a more precise solution.**

Explicit fixed point programming

- **There is a comprehensive array of 16 bit intrinsic functions.**
  (detailed in Visual DSP 4.0\Blackfin\include\ )
- **You must program using these explicitly.**
- **Intrinsics are 'inlined' and are not function calls**
- **Optimiser fully understands built-ins and their effects**

```
#include <fract.h>
fract32 fdot(fract16 *x, fract16 *y, int n)
{
   fract32 sum = 0;
   int i;
   for (i = 0; i < n; i++)
      sum = add_fr1x32(sum, mult_fr1x32(x[i], y[i]));
   return sum;
}
```

ETSI Builtins – fully optimised Fractional arithmetic to a standard specification

- **European Telecommunications Standards Institute's fract functions carefully mapped onto the compiler built-ins.**

add()	**sub()**	**abs_s()**	**shl()**
shr()	**mult()**	**mult_r()**	**negate()**
round()	**L_add()**	**L_sub()**	**L_abs()**
L_negate()	**L_shl()**	**L_shr()**	**L_mult()**
L_mac()	**L_msu()**	**saturate()**	**extract_h()**
extract_l()	**L_deposit_l()**	**L_deposit_h()**	**div_s()**
norm_s()	**norm_l()**	**L_Extract()**	**L_Comp()**
Mpy_32()	**Mpy_32_16()**		

- **Immediate optimisation of ETSI standard codecs.**
- **Highly recommended!**

Pointers or Arrays?

- **Arrays are easier to analyse.**

```
void va_ind(int a[], int b[], int out[], int n) {
        int i;
        for (i = 0; i < n; ++i)
                out[i] = a[i] + b[i];
}
```

- **Pointers are closer to the hardware.**

```
void va_ptr(int a[], int b[], int out[], int n) {
        int i;
        for (i = 0; i < n; ++i)
                *out++ = *a++ + *b++
}
```

- **Which produces the fastest code?**

- **Often no difference.**
- **Start using array notation as easier to understand.**
- **Array format can be better for alias analysis in helping to ensure no overlap.**
- **If performance is unsatisfactory try using pointers.**

Loops

- It is considered a good trade off to slow down loop prologue and epilogue to speed up loop.
- Make sure your program spends most of its time in the inner loop.
- The optimizer basically “works by unrolling loops”.
  - Vectorisation
  - Software pipelining

- What is software pipelining?
  - Reorganizing the loop in such a way that each iteration of software-pipelined code is made from instructions of different iterations of the original loop

Simple MAC loop:

load, multiply/accumulate, store

```
CYCLE                1   2    3    4    5    6 .....100
                    L1  M1   S1
                        L2   M2   S2
                             L3   M3   S3
                                  L4   M4   S4
                                    ……..……
```

Effects of Vectorisation and Software Pipelining on Blackfin

- **Simple code generation: 1 iteration in 3 instructions**

```
R0.L = W[I1++]
R1.L = W[I0++];
A1+= R0.L*R1.L;
```

- **Vectorised and unrolled once: 2 iterations in 3 instructions**

```
 R0 = [I1++]
R1 = [I0++]
A1+= R0.H*R1.H, A0+= R0.L*R1.L (IS)
```

- **Software pipeline: 2 iterations in 1 instruction**

```
R0 = [I1++] || R1= [I0++];

LSETUP (._P1L2 , ._P1L3-8) LC0=P1;
.align 8;

._P1L2:

A1+= R0.H*R1.H, A0+= R0.L*R1.L (IS) || R0 = [I1++] || R1= [I0++];
._P1L3:
A1+= R0.H*R1.H, A0+= R0.L*R1.L (IS);
```

Do not unroll inner loops yourself

- **Good - compiler unrolls to use both compute blocks.**

```
for (i = 0; i < n; ++i)
   c[i] = b[i] + a[i];
```

- **Bad - compiler less likely to optimise.**

```
for (i = 0; i < n; i+=2) {
   xb = b[i]; yb = b[i+1];
   xa = a[i]; ya = a[i+1];
   xc = xa + xb; yc = ya + yb;
   c[i] = xc; c[i+1] = yc;
}
```

- OK to unroll outer loops.

Do not software pipeline loops yourself.

The original loop (good)

```
float ss(float *a, float *b, int n){

float sum = 0.0f;
int i;
for (i = 0; i < n; i++)
{
sum += a[i] + b[i];
}
return sum;
}
```

A pipelined loop (bad)

```
float ss(float *a, float *b,
  int n ) {

  float ta, tb , sum = 0.0f;
  int i = 0;
  ta = a[i]; tb = b[i];
  for (i = 1; i < n; i++) {
    sum += ta + tb;
    ta = a[i]; tb = b[i];
  }
  sum += ta + tb;
  return sum;
}
```

Avoid loop carried dependencies

- **Bad: Scalar dependency.**

```
for (i = 0; i < n; ++i)  x = a[i] - x;
```

Value used from previous iteration. So iterations cannot be overlapped.

- **Bad: Array dependency.**

```
for (i = 0; i < n; ++i)    a[i] = b[i] * a[c[i]];
```

Value may be from previous iteration. So iterations cannot be overlapped.

- **Good: A Reduction.**

```
for (i = 0; i < n; ++i)    x = x + a[i];
```

Operation is associative. Iterations can be reordered to calculate the same result.

- **Good: Induction variables.**

```
for (i = 0; i < n; ++i)    a[i+4] = b[i] * a[i];
```

Addresses vary by a fixed amount on each iteration. Compiler can see there is no data dependence.

You can experiment with Loop structure

- **Unify inner and outer Loops.**
  - **May make loop too complex, but optimiser is better focused.**
- **Loop Inversion. - reverse nested loop order.**
- **Unify sequential loops**
  - **reduce memory accesses – can be crucial when dealing with external memory.**

– And remember you can code explicit built-ins to force vectorisation.

e.g.. fract2x16 mult_fr2x16(fract2x16 f1,fract2x16 f2)

Failure to engage Hardware Loop.

- **The LSETUP zero overhead hardware loop is very effective.**
- **And signals a well understood loop.**

- **Reasons for not getting the desired HW loop from the compiler include:**

(1) There are only two, so only the two deepest nested cycles.

(2) Calls to complex or unknown functions in the loop.

Calls to simple, loop-less functions are accepted if the function is completely visible to the compiler.

(3) There must be no transfer of control into the loop other than the normal entry.

- **And additional transfers of control out of the loop, while accepted, lower efficiency.**

Vectorisation

- **Simultaneous operations on a number of adjacent data elements.**
- **But, programs written in serial, one-at-a-time form**
  - **compiler finds opportunities for many-at-a-time**

Valuable on architectures with wide data paths or parallel operations

These Factors Make Vectorisation Difficult

- **Non-sequential Memory References**
  - **vectorisation and SIMD must be based on sequential data**
  - **sometimes data layout can be modified, or algorithm changed**
- **Uncertain alignment of data**
  - **memory references usually must be naturally aligned**
  - **pointers, function parameters might have any value**
- **Uncertain Iteration Counts**
  - **vectorised loops count in larger increments: 2, 4, 8**
  - **more code required to deal with run-time-known value**
- **Possible Aliasing of Data**
  - **Vectorisation cannot occur if there is a question about inputs and outputs affecting each other**

Word align your data

- 32-bit loads help keep compute units busy.
- 32-bit references must be at 4 byte boundaries.
- Top-level arrays are allocated on 4 byte boundaries.

- Only pass the address of first element of arrays.
- Write loops that process input arrays an element at a time.
- Nested loops - make sure inner loop is word aligned.

- The Blackfin does not support odd aligned data access. If the compiler thinks that your 2 byte or 4 byte data *might* stray across odd boundaries, then it will start planting alternate loop forms and testing code to handle misaligned access. While helpful, this is better avoided!

Think about the data layout.

- **Arrange for 32 bit loads** **Example: Complex Data.**
- **short Real_Part[N];**
- **short Imaginary_Part [N];**
- **Wrong! Two loads required.**
- **short Complex [N*2];**
- **One 32 bit load sufficient to load both parts.**

Construct Arrays with high visibility.

- **A common performance flaw is to construct a 2D array by a column of pointers to malloc'ed rows of data. This allows complete flexibility of row and column size and storage.**
- **But it kills the optimiser which no longer knows if one row follows another and can see no constant offset between the rows.**

Watch Memory Access Patterns
Facilitate 32 bit and sequential access.

- **Need to be careful how program sweeps thru memory**

Original form moves through memory jumping "NC' words at a time -- cannot vectorize

```
for (i=0; i<NC; i++) {      // bad
   for (j=0; j<NC; j++) {
     int sum = 0.0;
     for (k=0; k<NUM_SAMPS; k++)
         sum += Input[k*NC + i] * Input[k*NC + i];
     Cover[i*NC + j] = sum / NUM_SAMPS;
 }
}
```

```
for (i=0; i<NC; i++) {      // good
   for (j=0; j<NC; j++) {
     int sum = 0.0;
     for (k=0; k<NUM_SAMPS; k++)
         sum += Input[i*NC + k] * Input[i*NC + k];
     Cover[i*NC + j] = sum / NUM_SAMPS;
 }
}
```

Aliasing: A prime cause of poor performance

- Watch Pointers which come from outside :- **Arguments, globals.**
- Watch Pointers which serve several purposes.
  (alias analysis is “flow free”)
- Watch Implicit pointers **– Reference parameters – esp. arrays.**
  Some of this is experience. Look at any pointers.
  Some comes from code inspection: why were expected optimisations inhibited?
- **Avoid use of global variables in application code**
  - **Ultimately hurts Maintainability and Performance**
  - **Compiler optimisations are often blocked by global-ness**
    - Eg. Don't use global scalars inside loops (or as loop exit conditions), as the compiler can't tell if a write through an arbitrary pointer will clobber that global.

Inter-procedural Analysis : -ipa

To produce the best possible code for a function the optimiser needs information about:

- **Data alignment**
- **Possible argument interference**
- **Number of iterations**

First compilation uncovers information.

- **Compiler called again from link phase if procedures will benefit from re-optimisation.**
- **Inter-procedural alias analysis associates pointers with sets of variables they may point to.**
- **Weakness: A control-flow independent analysis.**

(context-free)

#pragma no_alias

- **#pragma no_alias**

```
for (i=0; i < n; i++)
    out[i] = a[i] + b[i];
```

- **No load or store in any iteration of the loop has the same address as any other load or store in any other iteration of the loop. If this is untrue, the program will give incorrect answers.**
- **Or use Qualifier restrict. “This pointer cannot create an alias”.**

#pragma vector_for

- **#pragma vector_for**

```
for (i=0; i<100; i++)
    a[i] = b[i];
```

- **The vector_for pragma notifies the optimiser that it is safe to execute two iterations of the loop in parallel.**

Array alignment on Blackfin

- __builtin_aligned(pointer,4); // aligned on word(32-bit) boundary.
- Executable statement, not #pragma, so when used on parameter arrays, must come after declarations in receiving function.

#pragma all_aligned

On Blackfin this asserts that the pointers are word aligned.

- Takes an optional argument (*n*) which can specify that the pointers are aligned after *n* iterations.

  #pragma all_aligned(1)

  would assert that, after one iteration, all the pointer induction variables of the loop are word aligned on Blackfin.

#pragma different_banks

- Asserts that every memory access in the loop goes to a different L1 memory bank.

#pragma loop_count

- example

```
int i;
#pragma loop_count(24,48,8)
for (i=0; i<n; i++)
  sum += a[i] * b[i];
```

- minimum trip count – used to decide to omit loop guards
- maximum trip count – used to decide it it is worth slowing down prologue and epilogue for a faster loop?
- trip modulo – used during software pipelining and vectorisation, does compiler need to worry about odd number of iterations?

Volatile is an important tool

- **Volatile is essential for hardware or interrupt-related data**
- **Some variables may be accessed by agents not visible to compiler.**
  - **accessed by interrupt routines**
  - **set or read by hardware devices**
- **'volatile' attribute forces all operations with that variable to be done exactly as written**
  - **variable is read from memory each time**
  - **variable is written back to memory each time**
  - **The exact order of events is preserved.**

  Optimiser must know the effect of each operation.

Missing a Volatile qualifier is the largest single cause of Support Requests now!

And the opposite "const" can help too.

- **Writing const short *p, instead of short *p, when p accesses const data is something that helps our alias analysis quite a lot.**

Circular addressing

- **A[i%n]**

The compiler now attempts to treat array references of the form array[i%n] as circular buffer operations.

-force-circbuf can help the compiler to accept the safety of a circular buffer access.

- **Explicit circular addressing of an array index:**

long circindex(long index, long incr, unsigned long nitems)

- **Explicit circular addressing on a pointer:**

*void * circptr(void *ptr, long incr, void *base, unsigned long buflen)*

Tricks that Compilers still miss

- **Automatic Bit reversed addressing**
- **Reduction of multiple statements of C to single complex instructions. Eg BitMux, Viterbi.**

Know your vocabulary.
Example: The count_ones intrinsic

- **Original Problem:**

```
U16 parityCheck(U32 dataWord[], const S32 nWords) {
 S32 i,j;   U32 accParity=0;
 for(j=0; j < nWords; j++){
  for (i=0; i < 32; i++){
      if (((dataWord[j] >> i) & 0x0001) == 1)
          accParity++; }}
 return((accParity & 0x00000001) ? 1 : 0);
}
```

- **Hardware has a special instruction to count bits.**

```
for(j=0; j < nWords; j++)
   accParity += count_ones(dataWord[j]);
return((accParity & 0x0001)?1:0);
```

Sixty times faster!

The inline asm()

- **ASMS() used to be a hard barrier to all optimisation.**
- **Now when the asm uses specific registers or touches memory directly. It can tell the compiler what it "clobbers". The fourth field in the asm statement specifies the "clobbered" registers, or "memory".**
  - **Notice that "memory" is a rather vague assertion.**
- **This is a big improvement, but an asm() is still dangerous and no match for a compiler intrinsic. The compiler understands intrinsics fully.**
- **Only use asm() when you need a very specific capability and avoid inner loops if you can.**

Example of inline asm.
A widening multiply.
(fract 32*32 using 16 bit operations).

```
inline int32_t MULT32(int32_t x,
 int32_t y) {
   ogg_int32_t hp;
   asm volatile (                    \
    " a1 = %1.l*%2.l(fu);            \n  \
      a1 = a1 >> 16;                 \n  \
      a1+= %1.h*%2.l(m,is);          \n  \
      a1+= %2.h*%1.l(m,is);          \n  \
      a1 = a1>>>16;                  \n  \
      %0 = (a1+= %1.h*%2.h)(is);":  \
      "=O"(hp):                      \
      "d"(x),"d"(y):                 \
      "a1", "astat");
   return hp;
}
```

And also for C functions: #pragma regs_clobbered *string*

Specify which registers are modified (or clobbered) by that function.

Tuning Control Code

Optimise Conditionals with PGO
"Profile Guided Optimisation" : -Pguide

Simulation produces execution trace.

- **Use the compiled simulator (hundreds of times faster) than conventional simulator.**
- **Problem: If what matters to you is worst case, not majority case, then choose training data appropriately.**

- **Then re-compile program using execution trace as guidance.**
  - **Compiler now knows result of all conditional operations.**

- **The payoff is significant: There are eight stalls to be saved.**
  - **The compiler does not aim for a static prediction – that still costs four stalls. The compiler goes all out for zero delay.**

- **The compiler will re-arrange everything to maximise drop through on conditional jumps. That's optimal – zero cycle delays. So your code will look completely different!**

Or Manual guidance

- **You can guide the compiler manually with:**

- **expected_true()**
- **expected_false().**

For instance,

```
If ( expected_false( failure status ) ) {
    < Call error routine >
}
```

Will lift very seldom executed error handling code out of main code stream. Execution will drop through to next item after testing condition.

Replace Conditionals with Min,Max,Abs.

Simple bounded decrement

```
k = k-1;
if (k <  -1)
   k = -1;
```

Programming "trick"

```
k = max (k-1, -1);
```

Avoid jump instruction latencies and simplifying control flow helps optimisation.

```
R0 +=  -1;
R1 =   -1;
R0 = MAX (R1,R0);
```

The compiler will often do this automatically for you, but not always in 16 bit cases.

BF ISA Note: Min and Max are for signed values only.

Removing conditionals 2

- **Duplicate small loops rather than have a conditional in a small loop.**

- **Example**

```
for {
        if  { ..... } else {.....}
    }

=>  if {
        for {.....}
    } else {
        for  {.....}
    }
```

Removing Conditionals 3
Predicated Instruction Support

- **The blackfin predicated instruction support takes the form of:**

  IF (CC) reg = reg.
- **Much faster than a conditional branch. (1 cycle) but limited.**
- **Recode to show the compiler the opportunity.**
- **For instance – consider speculative execution.**
  - **if (A) X = EXPR1 else X = EXPR2;**

  Becomes:
  - **X = EXPR1; IF (!A) X = EXPR2;**

  Or X=EXPR1; Y=EXPR2; if (!A) X=Y;

- **Generally,speculation is dangerous for compilers. Help it out.**
- **Eg. Compiler will not risk loading off the end of a buffer for fear of address errors unless you code –extra-loop-loads.**

Removing Conditionals 4

- **Is approach suitable for a pipelined machine**

```
sum = 0;
for (I=0; I<NN; I++) {
 if ( KeyArray[val1][10-k+I] == '1' )
    sum = sum + buffer[I+10]*64;
 else
    sum = sum -  buffer[I+10]*64;
}
```

- **Faster Solution removes conditional branch. Multiplication is fast: let KeyArray hold +64 or -64**

```
sum = 0;
for (I=0; I<NN; I++)
  sum += buffer[I+10] * KeyArray[val1][10-k+I];
```

- **Compiler is not able to make this kind of global change**

Avoid Division.

- There are no divide instructions – just supporting instructions.
- Floating or integer division is very costly
- Modulus(%) also implies division.
- Get Division out of loops wherever possible.

Classic trick: - power of 2 ➔ shift.

- Division by power of 2 rendered as right shift – very efficient.
- Unsigned Divisor – one cycle. (Division call costs 35 cycles)
- Signed Divisor – more expensive. (Could cast to unsigned?)

  x / 2^n = ((x<0) ? (x+2^n-1) : x) >> n // Consider –1/4 = 0!

- Example: signed int / 16

```
R3 = [I1];              // load divisor
CC = R3 <  0;           // check if negative
R1 =   15;              // add 2^n-1 to divisor
R2 = R3 + R1;
IF CC R3 = R2 ;         // if divisor negative use addition result
R3 >>>=  4;             // to the divide as a shift
```

- Ensure compiler has visibility. Divisor must be unambiguous.

There are two implementations of Division.

- **On a Blackfin, the cost depends on the range of the inputs.**

  (1) Can use Division primitives Cost ~ 40 cycles
  (Loosely speaking, the result and divisor fit in 16 bits.)

  (2) Bitwise 32 bit division Cost ~ 400 cycles

Consider numeric range when dividing.

Division can be created by For loops.

- **The Hardware loop constructs require an iteration count.**

- **for (I = start; I < finish; I += step)**

  compiler plants code to calculate:
  iterations = (finish-start) / step

Use the laws of Algebra

- **A customer benchmark compared ratios coded as:**

  if (X/Y > A/B)

Recode as:

  if (X * B > A * Y)

Another way to lose divisions!

- **Problem: possible overflow in fixed point.**

- **The compiler does not know anything about the real data precision. The programmer must decide. For instance two 12 bit precision inputs are quite safe. (24 bits max on multiplication.)**

Dreg or Preg?

- The Blackfin design saves time and power by associating registers with functional units. That splits the register set into Dregs and Pregs. But it costs three stalls to transfer from Dreg to Preg. So you want to avoid chopping back and forth.
- Avoid address arithmetic that is not supported by Pregs and you will avoid transfers from Preg to Dreg and back.
- For instance. Multiplication except by 2 or 4 has to be done outside the DAG. So indexing an odd sized struct array is OK using auto-increment, but arbitrary element addressing may involve the Dregs.

Convergent: DSP and micro-controller - Integer size?

- **16 bit for arrays of data in DSP style calculations.**
- **32 bits for “control” variables in micro-controller code.**
- **The ISA is not symmetric for 8, 16 and 32 bits.**
  - **Example: Comparison can only be done in 32 bits.**
  - **So 8 bit or 16 bit conditions have to be preceded by coercions, which also consume more registers. The compiler will optimise away as much of this as possible, but best is to make control variables 32 bits.**
  - **Especially For loop control variables.**

Varying multiplication cost.

- **32 bit multiplication takes three cycles.**
  - `R0 *= R1;`
- **You can do two 16 bit multiplications in one cycle.**
  - **And you can load and store in the same cycle.**
  - **That's up to six times the throughput.**
  - **Eg.**
```
A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS)
   || R0.L = W[I1++]
   || R0.H= W[I0++];
```
- **Be careful to specify 16 bit inputs and outputs to avoid unwanted 32 bit multiplication.**
  - **Address arithmetic is a common source of 32 bit mults.**

Function Inlining (-Oa) and The “inline” qualifier

- **Inlining functions (inserting body code at point of call) :**
  - **Saves call instruction (multi-cycle)**
  - **Saves return (multi-cycle)**
  - **Saves construction of parameters and return values.**
  - **Gives the optimiser greater visibility.**
  - **Gives the optimiser more instructions to schedule.**
- **But code size increases.**
- **Automatic: Tick Function inlining box. -Oa**
- **Manual: Mark small functions with the “inline” qualifier.**

Zero trip loops

- C definition of a FOR loop is Zero or more iterations.
- Blackfin HW loop is defined as One or more iterations.
- Result is guard code before loops jumping around if zero trip.

```
R7=[FP+ -8] || NOP;
CC = R7 <=  0
IF CC JUMP  ._P3L15 ;
    LSETUP and loop body.
    ....
._P3L15:
```

- This can be avoided by using constant bounds

  FOR (I=0; I<100; I++) not FOR (I=min; I<max; I++)

- Or with:

```
#pragma loop_count(1,100,1)
```

Blackfin Memory Performance

Evaluation tip: Watch the Bus speed.
Some good CPU/BUS speed settings for Blackfins

30 mhz clock

DF=1 (giving 15 mhz granularity)

Multiple	CPU Speed	Bus speed Div = 4	Bus speed Div = 5	Bus speed Div = 6
30	450	112.50	90.00	75.00
31	465	116.25	93.00	77.50
32	480	120.00	96.00	80.00
33	495	123.75	99.00	82.50
34	510	127.50	102.00	85.00
35	525	131.25	105.00	87.50
36	540		108.00	90.00
37	555		111.00	92.50
38	570		114.00	95.00
39	585		117.00	97.50
40	600		120.00	100.00
41	615		123.00	102.50
42	630		126.00	105.00
43	645		129.00	107.50
44	660		132.00	110.00
45	675			112.50
46	690			115.00
47	705			117.50
49	735			122.50
50	750			125.00

27 mhz clock

DF=1 (giving 13.5 mhz granularity)

Multiple	CPU Speed	Bus speed Div = 3	Bus speed Div = 4	Div = 5	Div = 6
29	392	130.50	97.88	78.30	65.25
30	405		101.25	81.00	67.50
31	419		104.63	83.70	69.75
32	432		108.00	86.40	72.00
33	446		111.38	89.10	74.25
34	459		114.75	91.80	76.50
35	473		118.13	94.50	78.75
36	486		121.50	97.20	81.00
37	500		124.88	99.90	83.25
38	513		128.25	102.60	85.50
39	527		131.63	105.30	87.75
40	540			108.00	90.00
41	554			110.70	92.25
42	567			113.40	94.50
43	581			116.10	96.75
44	594			118.80	99.00
45	608			121.50	101.25
46	621			124.20	103.50
47	635			126.90	105.75
49	662			132.30	110.25
50	675				112.50
51	689				114.75
52	702				117.00
53	716				119.25
54	729				121.50
55	743				123.75
56	756				126.00

Memory Hierarchy - Caching

- **The cache occupies some of the L1 memory. (it's optional) Recommend use caching from the start .**

- **Either use the Project Wizard or:**

(1) Use Linker option USE_CACHE

(2) Insert in the C program:

#pragma "retain_name"

A static integer "cplb_ctrl = 15"

(3) Write Back mode holds writes until cache line dismissed.

- **Requires a change to the CPLB table to activate.**

- **The alternative use of L1 as non-cached data is forced with the section directive.**

```
static section("L1_data") int X[100];
```

Spread across the SDRAM banks.

The ADSP-BF533 EBIU allows for 4 internal banks in an external SDRAM bank to be accessed simultaneously, reducing the stalls on access compared to keeping program and data in one bank.

- **An SDRAM row change can take 20-50 cycles depending on the SDRAM type.**
- **Macro "PARTITION_EZKIT_SDRAM" used with standard LDF files spreads out your application with code, heap and data in separate banks**
- **Results from EEMBC MPEG4 Encoder**

6.5% performance improvement.

(Executing code and data from external memory with caching off.)

SDRAM access – significant costs.
(using Bf533 EZkit)

Average time over 10,000 16 bit transfers				
		Sequential	Alternate Rows	Alt Rows/diff Banks
L1	Read	1	1	1
L3 (cached)	Read	**7.7**	**9.4**	**7.7**
L3	Read	**40.4**	**141.5**	
L1	Write	1	1	1
L3 (cached)	Write/WT	**5.2**	**45.7**	**5.2**
(Times in cycles at 600 mhz core speed)				

- **Generic problem to industry – much faster processors, slightly faster memories.**
- **SDRAM access can be more significant than all the cleverness the compiler may engage in vectorising a loop.**
- **Cache gives a three times performance improvement on first access because the data is requested and sent in 32 byte "lines". The big pay off with cache is in data re-use.**
- **Cache is reactive, not anticipatory like DMA.**

SDRAM row switching, slows you down by a factor of 3.5 for reading and nine! for writing.

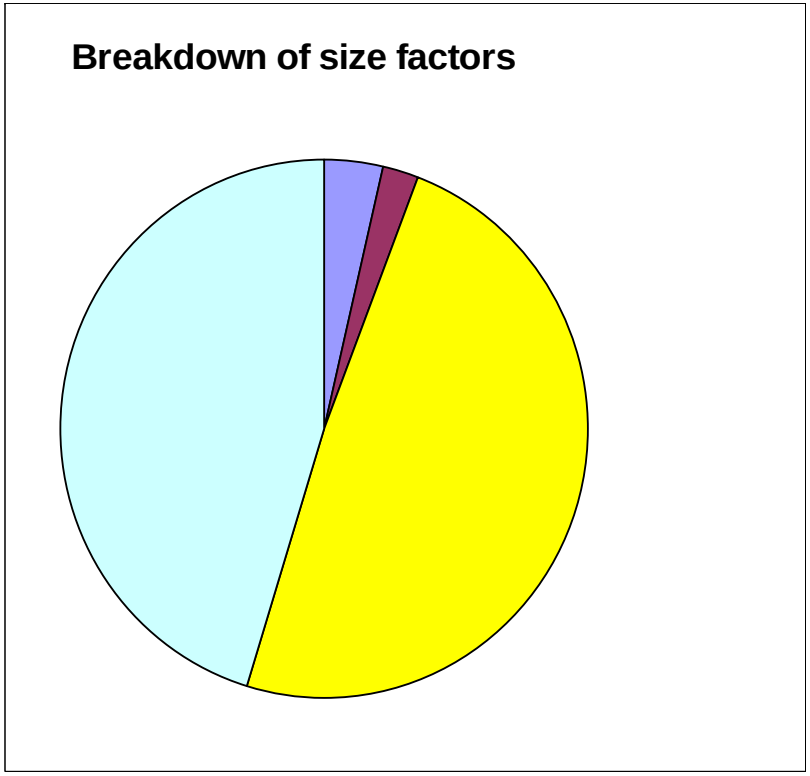
Code Speed versus Space

- **Compile for minimum code size or for maximum speed?**
- **Many optimisations increase the code size.**
- **This is at it's most extreme in DSP Kernels, where loop unrolling, vector prologs and epilogs increase space several times.**
- **But minimising space in DSP kernels would cost us far too much performance. So it becomes important to highlight code to be compiled for space or for speed.**
- **-Os and –O settings are available down to the function level.**

```
#pragma optimize_for_space
#pragma optimize_for_speed
```

Which Options? (code density suite)

- Control code size has significant variability on a Blackfin.
- As much as a 50% variation in code size can be available.
- No mode change calling between speed and space optimised code. Space optimised code has no restrictions on functionality.



- 2% -Ofp
- 48% -O / -Os
- 45%

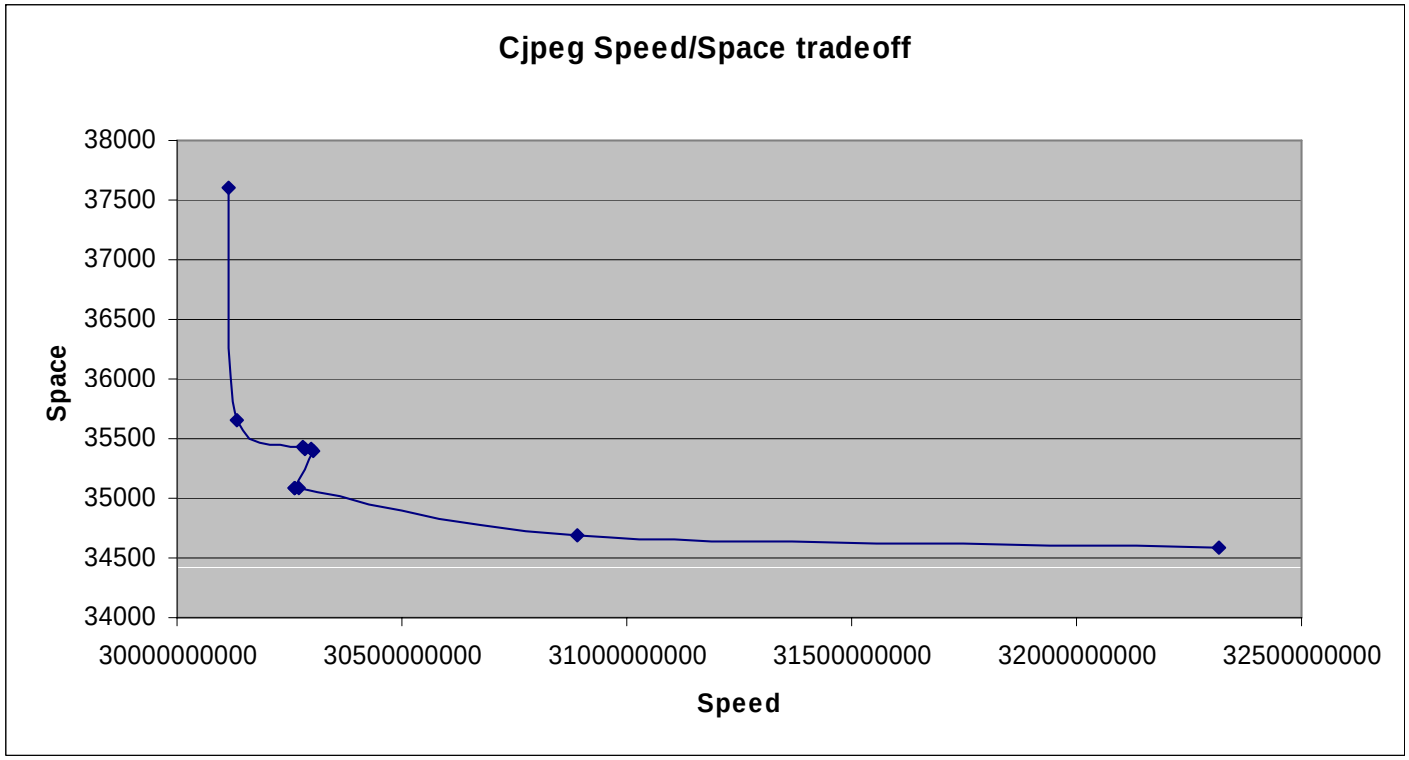
Function inlining.

4% misc.

Advanced Automatic speed/space blending.

- **-Ov (1 to 100)**
- **Based on Profile Guided Optimisation analysis.**
- **The compiler knows exactly how often each block of code is executed, which is a good guide to its importance.**
- **The compiler knows pretty well how size expansive each of its speed increasing transformations is.**
- **Combining this information, the compiler can apply the –OvN as a threshold and decide whether to (individually!) compile each block of the program for space of speed.**
- **This is staggeringly simple, and very effective.**

Trial results from –OvNum
(stepping "Num" from 0 to 100 by 10)



- **EEMBC Consumer. All five apps show these curves.**
- **Shows we have a large range offering reduced space.**
- **Shows Peak performance is at limited high range values.**

The Manual method of blending space and speed. – Try file level granularity.

Based on the EEMBC Consumer V1 Cjpeg benchmark.

	<=========SPEED=========>			<======SIZE=========>		
Filename	**iterations/sec**	**change**	**% of avail.**	**size**	**diff**	**% of avail**
StartPoint	20.19			38,060		
bm_lib.c	20.19	0.00	0.00%	38,052	8	0.28%
bmark_lite.c	20.19	0.00	0.00%	38,052	0	0.00%
cdjpeg.c	20.19	0.00	0.00%	38,052	0	0.00%
cjpeg.c	20.19	0.00	0.00%	38,052	0	0.00%
filedata.c	20.19	0.00	0.00%	38,028	24	0.83%
fileio.c	19.843	0.35	**24.95%**	38,028	0	0.00%
jcapmin.c	19.842	0.00	0.07%	37,860	168	5.79%
jcapistd.c	19.839	0.00	0.22%	37,820	40	1.38%
jccoeffct.c	19.833	0.01	0.43%	37,620	**200**	**6.89%**
jccolor.c	19.607	0.23	**16.25%**	37,316	**304**	**10.47%**
jcdctmgr.c	19.472	0.13	**9.71%**	37,284	32	1.10%
jchuff.c	19.225	0.25	**17.76%**	37,060	**224**	**7.71%**
jcinit.c	19.225	0.00	0.00%	37,052	8	0.28%
jcmainct	19.225	0.00	0.00%	37,004	48	1.65%
jcmarker.c	19.225	0.00	0.00%	36,772	**232**	**7.99%**
jcmaster.c	19.225	0.00	0.00%	36,620	152	5.23%
jcomapi.c	19.225	0.00	0.00%	36,580	40	1.38%
jcparam.c	19.225	0.00	0.00%	36,428	152	5.23%
jcprepct.c	19.222	0.00	0.22%	36,356	72	2.48%
jcsample.c	19.151	0.07	**5.10%**	36,212	144	4.96%
jdatdst.c	19.151	0.00	0.00%	36,180	32	1.10%
jerror.c	19.151	0.00	0.00%	36,164	16	0.55%
jfdctint.c	18.804	0.35	**24.95%**	**35,780**	**384**	**13.22%**
jmemansi.c	18.803	0.00	0.07%	35,740	40	1.38%
jmemmgr.c	18.802	0.00	0.07%	35,308	**432**	**14.88%**
jutils.c	18.8	0.00	0.14%	35,292	16	0.55%
rdbmp.c	18.8	0.00	0.00%	35,156	136	4.68%
str.c	18.799	0.00	0.07%	35,156	0	0.00%

- **Yellow files matter for speed.**
- **Blue files matter for space.**
- **22 out of 28 files are almost irrelevant for speed tradeoffs and could be compiled for min space.**
- **It's not always the same files that matter for speed and space.**

Correlate speed and space, filter and order the file list.

(These figures are percentages of the available variation in space and speed, not total application space and speed.)

- filedata.c -O (25% faster for no space increase)
- jccolor.c -O (9.7% faster for 1.1% space increase)
- jcdctmgr.c -O (17.76% faster for 7.7% space increase)
- jerror.c -O (25% faster for 13.2% space increase)
- jccoeffct.c -O (16.25% faster for 10.5% space increase)
- jcprepct.c -O (5% faster for 5% space increase)

- This method appears to give us some advantages.
  - We have removed most of the files from play.
  - We have interesting tradeoffs.

Examples

Some real examples based on programs submitted by customers and using the principles outlined in this presentation.

A real case of “complex” data structure – Before .....

```
for (INT16 j = 0; j < syncLength ; j ++)   // THIS LOOP VITAL TO APP PERFORMANCE.
{
        k = i + j;
        realSyncCorrelation += ((INT32)sample[k].iSample * syncCofs[j])  << 1 ;
        imagSyncCorrelation += ((INT32)sample[k].qSample * syncCofs[j])  << 1 ;
}
```

- **Producing:**

```
._P3L15:
    A0 += R1.L*R0.L (W32) || R3 = W[P1++P4] (X) || NOP;
    A1 += R3.L*R0.L (W32) || R1 = W[P1++] (X) || R0.L = W[I0++];
// end loop ._P3L15;
```

- **You must have an idea of what optimal code would look like.**
- **A vector MAC is desirable, but appears to be blocked by three load/store requests.**
- **Consider the Data layout.**

```
typedef struct
{
        INT16 iSample;      <<<<<<<< We pick up this, but have to assume only 2 byte alignment
        INT16 qSample;      <<<<<<<< We pick up this, but have to assume only 2 byte alignment
        INT16 agc;          <<<<<<<< This element is interspersed with the data we want.
}rxDMASampleStruct_t;       <<<<< 6 byte struct.  Odd lengths can be trouble.
```

... and After

- **Copying data to a temp struct is cheap compared to gains.**

```
INT16 sample_iq[287*2];  // local copy of data  - real and imaginary shorts consecutive.
   for (INT16 j = 0; j < syncLength ; j ++)
    {
      k = i + j;
      realSyncCorrelation += ((INT32)sample_iq[(k*2)] * syncCofs[j])  << 1 ;
      imagSyncCorrelation += ((INT32)sample_iq[(k*2)+1] * syncCofs[j])  << 1 ;
    }
```

- **Producing optimal code for a single cycle loop:**

```
._P3L15:
  A1+= R1.H*R3.L, A0+= R1.L*R3.L || R1 = [P0++] || R3.L = W[I0++];
// end loop ._P3L15;
```

- **Note the (k*2) form. This is a particularly strong way of expressing to the compiler that the alignment will be 4 bytes.**

Customer's Goal : get ported C to run as required

- **This is abstracted from a real evaluation of the Blackfin.**
- **Strategy for application acceleration:**

(1) Insert timing points and checking code. (50 million cycles)
(2) Optimisation on. (9.5 million cycles)
(3) Cache on. (1.4 million cycles)
(3) Investigate global optimisations.
(4) Profile
(5) Focus on hot spots
(6) Set exact MHZ and bus speed.
(7) Memory tuning for target device.

Problem #1 : Profile and Focus - ETSI

- **Statistical Profiler reveals ETSI fractional arithmetic costly. Code inspection reveals implemented by function calls.**
- **Engage compiler builtins for ETSI. This only requires a pre-processor setting**
- **#define ETSI_SOURCE**
- **#include <libetsi.h>**

ETSI call mostly collapse to mostly single machine instructions.

Result: Subsection reduced from 187,000 cycles to 141,000.

Problem #2: Conditional jumps found in critical loop

```
short corr_max(DATA *corr_up,DATA *corr_down,DATA *store,DATA gate_level, ushort nx)
{
    int j;
    for (j=0;j<len;j++) {
                        DATA maxval;                                    //    DATA is 16 bits.

                        maxval = *corr_up++;
                        if(*corr_down>maxval) maxval = *corr_down;
                        corr_down++;
                        if(maxval>gate_level)
                        {
                                if(maxval>*store) *store = maxval;
                        }
                        store++;
    }
    return 0;
}
```

- **What's wrong with this?**
- **Three conditional jumps inside a loop!**
- **Initial cost 67,000 cycles for this subsection, mostly jump stalls.**

Remove first Conditional jump!

- **Transform:**

```
maxval = *corr_up++;
if(*corr_down>maxval) maxval = *corr_down;
corr_down++;
```

to:

```
maxval = max(*corr_down++,*corr_up++);
```

- **Time cut from 67,000 to 40,000 cycles.**

- **Why did the compiler not perform this transformation?**
- **Wrong DATA type. Compiler does 32 bit values automatically, but is more cautious with 16 bit.**

Remove second Conditional jump

- **Look at the innermost conditional**

```
if(maxval>*store) *store = maxval;
```

- **Use MAX again and force condition to always execute.**

```
*store = max(*store,maxval);
```

- **A compiler will not do this for you. We are forcing an extra write, which the compiler will reckon is outside of its competence and might be counter productive. This is a programmer level decision.**

- **Saves another 6,500 cycles.**

Remove third and final Conditional jump

Facilitate a translation to " IF CC Rx = Ry.
A one cycle predicated instruction. Get rid of the jump, NOT the condition.

```
for (j=0;j<len;j++) {
                    unsigned int maxval;

                    oldvalue   = *store;
                    maxval     = max(*corr_down++,*corr_up++);
                    maxstore = max(oldvalue,maxval);
                    if(maxval>gate_level)                    //<<<<< Pattern for predicated instruction
                    {
                                        newvalue = maxstore;
                    } else {
                                        newvalue = oldvalue;
                    }
                    *store++ = newvalue;
}
```

The Problem is that we now write to "*store" unconditionally.
The compiler will not do this for you - the compiler will say "what if one of those addresses was invalid –or the programmer wanted to access memory strictly by "(maxval>gate_level)".
Quite right. Programmer level decision.

- **Result: Down another 16,500 cycles**

Remove coercion

- Sharp eyes will have noted that variable Maxval changed from a DATA (2 bytes) to a 4 byte variable.
- This is to remove an unnecessary coercion. (16 -> 32 bits)
- The problem is that the Blackfin control code style, including comparison operations is set up for 32 bits.
- Solution: Make maxval an unsigned integer.
- 2,000 cycles saved. (One instruction from the loop.)

Performance impact by removing All three CONDITIONAL LOOPS

- **The final machine code looks like this:**

```
                LSETUP (._P1L1 , ._P1L2-2) LC0=P5;
._P1L1:
                R0 = W[P0++] (X);
                R2 = W[P1++] (X);
                R0 = MAX (R0,R2) || R2=W[P2+ 0] (X) || NOP;   //<<<<< IF (1) => MAX instr
                CC = R1 < R0 (IU);
                R0 = MAX (R0,R2);            //<<<<<<<<<< IF (2) => MAX instruction
                IF !CC R0 = R2 ;             //<<<<<<<<<< IF (3) => Predicated instruction
                W[P2++] = R0;
                // end loop ._P1L1;
```

- **This is nearly optimal. Software pipelining can be triggered by prefacing the loop with #pragma no_alias.**

- **The loop is now running about 3.5 times faster than the original. There are no jump stalls.**

Problem #3: Opportunities with Memcpy

- **Statistical profile shows that we spend time in memcpy().**
- **Experimentally engage Write Back mode in the cache.**
- **This means that data will not be copied back to external memory until the cache line is retired.**

- **Setup in cplbtab.h.**
  - **#define CPLB_DWBCACHE CPLB_DNOCACHE | CPLB_CACHE_ENABLE**
  - **0x00000000, (PAGE_SIZE_4MB | CPLB_DWBCACHE),**

- **Subsection time reduces from 87,000 cycles to 50,000.**

Additional Information about Memcpy

- **Memcpy() is written to reward 4 byte alignment of data.**
  It checks word alignment and uses a word at a time loop if it can. Otherwise a byte by byte copy is performed.

- **Because: A Blackfin can do a 4 byte load and a 4 byte store in a single cycle, but you can only load or store a single byte per cycle.**

  Gives an Eight times advantage to 4 byte word operations.

- **Future work on this program might include the re-alignment of application data buffers.**

Problem #4: Loop Splitting?

```
for (k=0; k<numcols; k++)
    {
            kr1+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef1++);
            ki1+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef1++);
            kr2+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef2++);
            ki2+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef2++);
            matdc_val_loc++;
    }
```

The Blackfin has dual accumulators, but the compiler is only using one in processing this code.

The problem may be complexity or too many active pointers whose range is not obvious to the compiler (ie. An aliasing problem.)

Consider: Split into two simpler loops.

Impact of Loop Splitting

```
const int numcols=*matdc_numcols_loc++;
for (k=0; k<numcols; k++) {
        kr1+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef1++);
        ki1+=(LDATA)L_mult((fract16)*matdc_val_loc, (fract16)*f_coef1++);
        matdc_val_loc++;
}

for (k=0; k<numcols; k++) {
        kr2+=(LDATA)L_mult((fract16)*matdc_val_loc2, (fract16)*f_coef2++);
        ki2+=(LDATA)L_mult((fract16)*matdc_val_loc2, (fract16)*f_coef2++);
        matdc_val_loc2++;
}
```

This produces optimal code for each loop, using both accumulators.

```
._P1L7:
        MNOP || R0.H= W[I0++] || R1.L = W[I1++];
        A1+= R0.H*R1.L, A0+= R0.L*R1.L || NOP || R0.L = W[I0++];
        // end loop ._P1L7;
```

- **20,000 cycles saved.**

Next Step: Tune to final Platform

- The same techniques, applied in other areas reduced the time by another 40,000 cycles.
- We have come from 1.45 million to 0.73 million cycles in all.
- There remained lots of scope for #pragma no_alias but that required customer knowledge to do safely.
- The app is now moved to a BF532 from the BF533 development platform.
- This means less L1 memory and half the cache size.
- Time goes back up from 727,000 to 900,000 cycles.
- Exploration with profiler and cache monitor confirms this application is sensitive to memory bus activity.

Consider Bus Speed : Set Sclk and Cclk

- **The processor speed and bus speed of the Blackfin are variable.**

  Experimentation reveals that:
- **CCLK= 391.5MHz**
- **SCLK= 130.5MHz**

  Are optimal.

- **Explanation: The clock on the BF532 Ezkit is 27 mhz. This is multiplied by 29 and divided by 2 to give a mhz of 391.5. The bus speed is established by dividing this by three = 130.5 mhz.**
- **We were at 900,000 cycles, now we are at 839,000 cycles.**

Consider L1 SRAM allocation

- The default LDF file uses any free L1 memory and cascades the rest of the application to external memory.
- Study cache behaviour with Cache Viewer. A number of cache events are seen that represent collisions.
- Track these back to user data constructs.
- Hint: Noticed in statistical viewer a store instruction taking 12 times longer than adjacent instructions. – has to be memory problem.
- Placed user data causing cache collisions into L1 memory.
- Final speedup from 839,000 to 747,000 cycles.

End of Project.

Additional Information

- **Reference material**
  - **EE197 Blackfin Multi-cycle instructions and latencies.**
  - **Blackfin Compiler manual, chapter 2: Achieving Optimal Performance from C/C++ Source Code.**
  - **ETSI – European Telecommunications Standards Institute.**

- **For Questions, click the “Ask A Question” button**

  Or email to Processor.support@analog.com