

# CERN Snippets Dissected

Gelato ICE

16<sup>th</sup> April 2007

J.M. Dana



- We have millions of lines of “real” code
- Is our code optimized?
- How do compilers optimize our snippets?
- Which problems should we solve manually by simply changing the code?

- Remove data dependencies
- Loop unrolling
- Inlining
- Good (intelligent) memory management
- Improved branching ratios
- Speculation
- Parallelization
- etc.

- Our testbed:
  - 4 x 1.6 GHz Montecito
  - 16 GB RAM
  - Scientific Linux CERN 4.4 ia64
  
- Compilers used:
  - gcc-4.2-20070307
  - gcc-4.3-20070302
  - icc-9.0
  - icc-9.1-046
  - icc-10.0 (still confidential, so not used here)
  
- All snippets compiled using `-O2` (unless noted):
  - Standard optimization level used by CERN programmers
  
- Compiler generated assembly not analyzed (yet)!

- **GEANT4:**
  - `G4AffineTransform::InverseProduct`
- **CLHEP:**
  - `RanluxEngine::flat`
- **ROOT:**
  - `TRandom::Landau`
  - `TGeoArb8::Contains`
  - `TBuffer::ReadFastArrayDouble32`
  - `TGeoCone::Contains`

- It's a simple (for humans) geometric rotation and translation
- However, It seems that it's not so simple for compilers
- There is no RAW (Read After Write) dependencies, there is no loops so... could be an object related problem?

```
inline G4AffineTransform&
G4AffineTransform::InverseProduct( const G4AffineTransform& tf1,
                                   const G4AffineTransform& tf2)
{
    G4double itf2tx = - tf2.tx*tf2.rxx - tf2.ty*tf2.rxy - tf2.tz*tf2.rxz;
    G4double itf2ty = - tf2.tx*tf2.ryx - tf2.ty*tf2.ryy - tf2.tz*tf2.ryz;
    G4double itf2tz = - tf2.tx*tf2.rzx - tf2.ty*tf2.rzy - tf2.tz*tf2.rzz;

    rxx=tf1.rxx*tf2.rxx+tf1.rxy*tf2.rxy+tf1.rxz*tf2.rxz;
    rxy=tf1.rxx*tf2.ryx+tf1.rxy*tf2.ryy+tf1.rxz*tf2.ryz;
    rxz=tf1.rxx*tf2.rzx+tf1.rxy*tf2.rzy+tf1.rxz*tf2.rzz;

    ryx=tf1.ryx*tf2.rxx+tf1.ryy*tf2.rxy+tf1.ryz*tf2.rxz;
    ryy=tf1.ryx*tf2.ryx+tf1.ryy*tf2.ryy+tf1.ryz*tf2.ryz;
    ryz=tf1.ryx*tf2.rzx+tf1.ryy*tf2.rzy+tf1.ryz*tf2.rzz;

    rzx=tf1.rzx*tf2.rxx+tf1.rzy*tf2.rxy+tf1.rzz*tf2.rxz;
    rzy=tf1.rzx*tf2.ryx+tf1.rzy*tf2.ryy+tf1.rzz*tf2.ryz;
    rzz=tf1.rzx*tf2.rzx+tf1.rzy*tf2.rzy+tf1.rzz*tf2.rzz;

    tx=tf1.tx*tf2.rxx+tf1.ty*tf2.rxy+tf1.tz*tf2.rxz+itf2tx;
    ty=tf1.tx*tf2.ryx+tf1.ty*tf2.ryy+tf1.tz*tf2.ryz+itf2ty;
    tz=tf1.tx*tf2.rzx+tf1.ty*tf2.rzy+tf1.tz*tf2.rzz+itf2tz;

    return *this;
}
```

- Let me see what happens if I “neutralize” the objects

```

inline G4AffineTransform&
G4AffineTransform::InverseProduct(
    const G4AffineTransform& tf1,
    const G4AffineTransform& tf2)
{
    G4double rxx1=tf1.rxx;
    G4double rxy1=tf1.rxy;
    G4double rxz1=tf1.rxz;
    G4double ryx1=tf1.ryx;
    G4double ryy1=tf1.ryy;
    G4double ryz1=tf1.ryz;
    G4double rzx1=tf1.rzx;
    G4double rzy1=tf1.rzy;
    G4double rzz1=tf1.rzz;
    G4double tx1=tf1.tx;
    G4double ty1=tf1.ty;
    G4double tz1=tf1.tz;

    G4double rxx2=tf2.rxx;
    G4double rxy2=tf2.rxy;
    G4double rxz2=tf2.rxz;
    G4double ryx2=tf2.ryx;
    G4double ryy2=tf2.ryy;
    G4double ryz2=tf2.ryz;
    G4double rzx2=tf2.rzx;
    G4double rzy2=tf2.rzy;
    G4double rzz2=tf2.rzz;
    G4double tx2=tf2.tx;
    G4double ty2=tf2.ty;
    G4double tz2=tf2.tz;

    G4double itf2tx = - tx2*rxx2 - ty2*rxy2 - tz2*rxz2;
    G4double itf2ty = - tx2*ryx2 - ty2*ryy2 - tz2*ryz2;
    G4double itf2tz = - tx2*rxz2 - ty2*rzy2 - tz2*rzz2;

    rxx=rxx1*rxx2+rxy1*rxy2+rxz1*rxz2;
    rxy=rxx1*ryx2+rxy1*ryy2+rxz1*ryz2;
    rxz=rxx1*rzx2+rxy1*rzy2+rxz1*rzz2;

    ryx=ryx1*rxx2+ryy1*rxy2+ryz1*rxz2;
    ryy=ryx1*ryx2+ryy1*ryy2+ryz1*ryz2;
    ryz=ryx1*rzx2+ryy1*rzy2+ryz1*rzz2;

    rzx=rzx1*rxx2+rzy1*rxy2+rzz1*rxz2;
    rzy=rzx1*ryx2+rzy1*ryy2+rzz1*ryz2;
    rzz=rzx1*rzx2+rzy1*rzy2+rzz1*rzz2;

    tx=tx1*rxx2+ty1*rxy2+tz1*rxz2+itf2tx;
    ty=tx1*ryx2+ty1*ryy2+tz1*ryz2+itf2ty;
    tz=tx1*rzx2+ty1*rzy2+tz1*rzz2+itf2tz;

    return *this;
}

```

	Original	Optimized	Improvement
<b>gcc-4.2</b>	100.45	51.87	1.936
<b>gcc-4.3</b>	100.45	51.86	1.936
<b>icc-9.0</b>	156.88	43.62	3.596
<b>icc-9.1</b>	62.58	55.03	1.137

- icc-9.0 has the worst result without optimization... but it's the best one with our changes!
- On the contrary, icc-9.1 has the best result with the original code and the worst one with the optimization. Have they already solved the problem?
- gcc far behind icc-9.1 on the original code

- It's a random number generator with an important RAW dependency
- “carry” could be the bottleneck of our execution path

```
for( i = 0; i != nskip ; i++){  
    uni = float_seed_table[j_lag] - float_seed_table[i_lag] - carry;  
    if(uni < 0. ){  
        uni += 1.0;  
        carry = mantissa_bit_24;  
    }else{  
        carry = 0.;  
    }  
    float_seed_table[i_lag] = uni;  
  
    i_lag --;  
    j_lag --;  
    if(i_lag < 0) i_lag = 23;  
    if(j_lag < 0) j_lag = 23;  
}
```

- We have simplified the RAW dependency removing “carry” variable

```
bool carryB=(carry==mantissa_bit_24);  
count24 = 0;  
for( i = 0; i != nskip ; i++){  
    uni = float_seed_table[j_lag] - float_seed_table[i_lag];  
  
    if(carryB)  
        uni-=mantissa_bit_24;  
  
    if(uni < 0. ){  
        uni += 1.0;  
        carryB = true;  
    }else{  
        carryB = false;  
    }  
  
    float_seed_table[i_lag] = uni;  
  
    i_lag --;  
    j_lag --;  
    if(i_lag < 0)i_lag = 23;  
    if(j_lag < 0) j_lag = 23;  
}
```

	Original	Optimized	Improvement
<b>gcc-4.2</b>	73.03	56.00	1.30
<b>gcc-4.3</b>	85.17	59.55	1.43
<b>icc-9.0</b>	24.16	27.15	0.89
<b>icc-9.1</b>	49.31	39.02	1.26

- icc-9.0 doesn't need optimization (it's better with the original snippet!)
- What have they changed in icc-9.1?
- Very bad results for gcc!

- It's a random number generator with an static array of 982 Double\_t elements (8 bytes per element in our testbed)
- There are several possibilities for initializing this array

```
Double_t TRandom::Landau(Double_t mpv, Double_t sigma)
{
// Generate a random number following a Landau distribution
// with mpv(most probable value) and sigma
// Converted by Rene Brun from CERNLIB routine ranlan(G110)

#ifdef ADDCONST
const
#endif
Double_t ff[982] = {
0,0,0,0,0,-2.244733,
-2.204365,-2.168163,-2.135219,-2.104898,-2.076740,-2.050397,
-2.025605,-2.002150,-1.979866,-1.958612,-1.938275,-1.918760,
-1.899984,-1.881879,-1.864385,-1.847451,-1.831030,-1.815083,
-1.799574,-1.784473,-1.769751,-1.755383,-1.741346,-1.727620,
-1.714187,-1.701029,-1.688130,-1.675477,-1.663057,-1.650858,
-1.638868,-1.627078,-1.615477,-1.604058,-1.592811,-1.581729,
-1.570806,-1.560034,-1.549407,-1.538919,-1.528565,-1.518339,
.....
}
```

	<b>-O2</b>	<b>-O3</b>	<b>-O2 -DADDCONST</b>	<b>-O3 -DADDCONST</b>
<b>gcc-4.2</b>	21.36	21.34	3.96	3.92
<b>gcc-4.3</b>	21.34	21.34	3.92	3.92
<b>icc-9.0</b>	21.11	32.18	21.11	32.20
<b>icc-9.1</b>	24.55	24.53	24.55	24.53

- Improvement given by “const” flag is around 5.4 (gcc only) !!!
- Amazing results for gcc (better if it could check that array is RONLY without human intervention). gcc-3 was not so clever!
- -O3 is not a good flag for icc-9.0 (what is it trying to do?)

- This snippet presents several challenges:
  - Speculative computations
  - Inlining
  - Floating-point calculations optimization
  - Loop-unroll (x4)

```

Bool_t TGeoArb8::Contains(Double_t *point) const
{
  if (TMath::Abs(point[2]) > fDz) return kFALSE;
  // compute intersection between Z plane containing point and
  the arb8
  Double_t poly[8];
  Double_t cf = 0.5*(fDz-point[2])/fDz;
  Int_t i;

  for (i=0; i<4; i++) {
    poly[2*i]   = fXY[i+4][0]+cf*(fXY[i][0]-fXY[i+4][0]);
    poly[2*i+1] = fXY[i+4][1]+cf*(fXY[i][1]-fXY[i+4][1]);
  }
  return InsidePolygon(point[0],point[1],poly);
}

```

```

Bool_t TGeoArb8::InsidePolygon(Double_t x, Double_t y,
Double_t *pts)
{
  // Find if a point in XY plane is inside the polygon defines by PTS.
  Int_t i,j;
  Double_t x1,y1,x2,y2;
  Double_t cross;
  for (i=0; i<4; i++) {
    j = (i+1)%4;
    x1 = pts[i<<1];
    y1 = pts[(i<<1)+1];
    x2 = pts[j<<1];
    y2 = pts[(j<<1)+1];
    cross = (x-x1)*(y2-y1)-(y-y1)*(x2-x1);
    if (cross<0) return kFALSE;
  }
  return kTRUE;
}

```

- We have manually inlined “InsidePolygon” function and fully unrolled the loops
- It looks horrible now but, let us see how it works...

```

Bool_t TGeoArb8::Contains(Double_t *point) const
{
  if (TMath::Abs(point[2]) > fDz) return kFALSE;
  // compute intersection between Z plane containing point and the arb8
  Double_t poly[8];
  Double_t x1,y1,x2,y2,cross,x,y;
  Double_t cf = 0.5*(fDz-point[2])/fDz;
  Int_t i;

  x=point[0]; y=point[1];
  poly[0] = fXY[4][0] +cf*(fXY[0][0]-fXY[4][0]);
  poly[1] = fXY[4][1] +cf*(fXY[0][1]-fXY[4][1]);
  poly[2] = fXY[5][0] +cf*(fXY[1][0]-fXY[5][0]);
  poly[3] = fXY[5][1] +cf*(fXY[1][1]-fXY[5][1]);
  poly[4] = fXY[6][0] +cf*(fXY[2][0]-fXY[6][0]);
  poly[5] = fXY[6][1] +cf*(fXY[2][1]-fXY[6][1]);
  poly[6] = fXY[7][0] +cf*(fXY[3][0]-fXY[7][0]);
  poly[7] = fXY[7][1] +cf*(fXY[3][1]-fXY[7][1]);
  x1 = poly[0]; y1 = poly[1]; x2 = poly[2]; y2 = poly[3];
  cross = (x-x1)*(y2-y1) - (y-y1)*(x2-x1);
  if (cross<0) return kFALSE;
  x1 = poly[2]; y1 = poly[3]; x2 = poly[4]; y2 = poly[5];
  cross = (x-x1)*(y2-y1) - (y-y1)*(x2-x1);
  if (cross<0) return kFALSE;
  x1 = poly[4]; y1 = poly[5]; x2 = poly[6]; y2 = poly[7];
  cross = (x-x1)*(y2-y1) - (y-y1)*(x2-x1);
  if (cross<0) return kFALSE;
  x1 = poly[6]; y1 = poly[7]; x2 = poly[0]; y2 = poly[1];
  cross = (x-x1)*(y2-y1) - (y-y1)*(x2-x1);
  if (cross<0) return kFALSE;
  return kTRUE;
}

```

	Original -O2	Original -O3	Optimized -O2	Optimized -O3
<b>gcc-4.2</b>	89.30	47.57	46.74	46.81
<b>gcc-4.3</b>	89.60	48.48	46.79	46.78
<b>icc-9.0</b>	55.67	54.69	27.97	25.44
<b>icc-9.1</b>	66.70	64.15	27.03	24.50

- *gcc*: The algorithm is around 1.9 times faster using loop unrolling + inlining. It's something that -O3 also solves. We have two different possibilities with gcc (could I have it for -O2?)
- *icc*: -O3 doesn't help for icc. In short, if we don't use "manual" modifications icc can't optimize this snippet. However, if we use suggested modifications icc obtains the best results
- Next aim: get icc to perform the optimization automatically (both -O2 and -O3)

# TBuffer::ReadFastArrayDouble32

- This algorithm reads elements from a buffer and, if necessary, swaps their bytes

```
void TBuffer::ReadFastArrayDouble32(Double_t *d, Int_t n,
TStreamerElement *ele)
{
    if (n <= 0 || 4*n > fBufSize){
        printf("Read primer if \n");
        return;
    }

    if (ele && ele->GetFactor() != 0) {
        printf("Read primer if2 \n");
        Double_t xmin = ele->GetXmin();
        Double_t factor = ele->GetFactor();
        for (int j=0;j < n; j++) {
            printf("Read for elemento %d \n",j);
            UInt_t aint;
            *this >> aint;
            d[j] = (Double_t)(aint/factor + xmin);
        }
    } else {
        Float_t afloat;
        for (int i = 0; i < n; i++) {
            frombuf(fBufCur, &afloat);
            d[i]=afloat;
        }
    }
}
```

```
inline void frombuf(char *&buf, Float_t *x)
{
    #ifdef R__BYTESWAP
    #if defined(R__USEASMSWAP)
        union {
            volatile UInt_t i;
            volatile Float_t f;
        } u;
        u.i = Rbswap_32(*((UInt_t *)buf));
        *x = u.f;
    #else
        union {
            volatile char c[4];
            volatile Float_t f;
        } u;
        u.c[0] = buf[3];
        u.c[1] = buf[2];
        u.c[2] = buf[1];
        u.c[3] = buf[0];
        *x = u.f;
    #endif
    #else
        memcpy(x, buf, sizeof(Float_t));
    #endif
    buf += sizeof(Float_t);
}
```

- Executed by gcc.
- Executed by icc.

	<b>-O0</b>	<b>-O1</b>	<b>-O2</b>	<b>-O3</b>
<b>gcc-4.2</b>	157.56	43.71	FAILS!	FAILS!
<b>gcc-4.3</b>	157.59	43.85	FAILS!	FAILS!
<b>icc-9.0</b>	416.09	111.46	111.49	111.47
<b>icc-9.1</b>	412.14	119.07	119.33	119.32

- Using different algorithms (assembly based for gcc and C for icc). It's the original version
- gcc is around 2.6 times faster... but it fails with -O2 and -O3!!! (-O2 should be safe)

	<b>-O0</b>	<b>-O1</b>	<b>-O2</b>	<b>-O3</b>
<b>gcc-4.2</b>	200.74	51.27	FAILS!	FAILS!
<b>gcc-4.3</b>	200.71	51.44	FAILS!	FAILS!
<b>icc-9.0</b>	416.09	111.46	111.49	111.47
<b>icc-9.1</b>	412.14	119.07	119.33	119.32

- Using the same algorithm (without assembly for gcc)
- gcc is around 2.1 times faster than icc (assembly is better)... but it has failed with -O2 and -O3 again!!!

- This function returns true if point is inside a cone
- We're doing  $0.5/fDz$  twice, are compilers exploiting parallelism?
- Our small hack:  $rfDz=0.5/fDz$  (stored in the object)

```
Bool_t TGeoCone::Contains(Double_t *point) const
{
// test if point is inside this cone
  if (TMath::Abs(point[2]) > fDz) return kFALSE;
#ifdef NODIVIDE
  Double_t r2 = point[0]*point[0]+point[1]*point[1];
  Double_t rl = (fRmin2*(point[2]+fDz)+fRmin1*(fDz-point[2]))*rfDz;
  Double_t rh = (fRmax2*(point[2]+fDz)+fRmax1*(fDz-point[2]))*rfDz;
#else
  Double_t r2 = point[0]*point[0]+point[1]*point[1];
  Double_t rl = 0.5*(fRmin2*(point[2]+fDz)+fRmin1*(fDz-point[2]))/fDz;
  Double_t rh = 0.5*(fRmax2*(point[2]+fDz)+fRmax1*(fDz-point[2]))/fDz;
#endif
  if ((r2<rl*rl) || (r2>rh*rh)) return kFALSE;
  return kTRUE;
}
```

- Optimized.
- Original.

	<b>Original</b>	<b>Optimized</b>	<b>Improvement</b>
<b>gcc-4.2</b>	86.52	58.35	1.48
<b>gcc-4.3</b>	86.52	58.35	1.48
<b>icc-9.0</b>	64.94	47.50	1.36
<b>icc-9.1</b>	66.84	49.40	1.36

- It seems that eliminating divisions is important after all
- Around 1.4 faster
- icc better than gcc in all cases

- Sometimes, compilers can't optimize properly so we have two possibilities:
  - Teach programmers all around the world everything about optimization
  - Optimize compilers themselves adding better memory disambiguation, inlining heuristics, pattern recognition, speculation, parallelism, etc.
- icc has some problems with memory management (Landau). However, it seems better solving RAW dependencies (Ranlux)
- In most cases gcc is way behind (but not always!)
- -O2 should be safe in gcc (ReadFastArrayDouble32). However, the results with -O0 and -O1 are good!

# Q & A

Jose.Dana@cern.ch