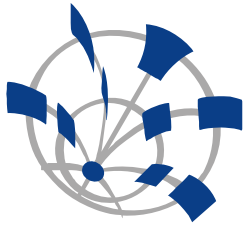
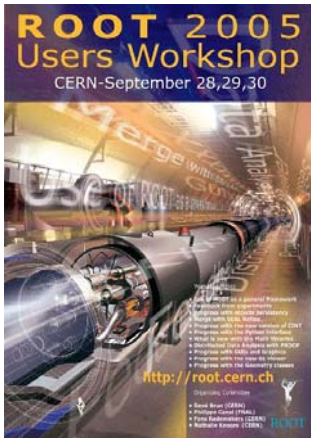
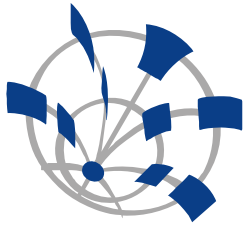


CERN openlab project



The value of snippets of HEP codes

ROOT 2005 – 30 September 2005
Sverre Jarp
with assistance from José Dana (Summer Student 2005)
IT Division – CERN

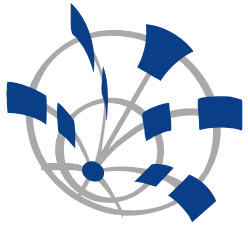


What is the issue?

- The profiles of HEP programs are re… …ely flat.
  - Without single "hot spots" where it i… …icient to verify if "optimal" code generation takes …e.
- Remedy:
  - Isolate solicited CPU-i… …sive C++ classes/methods (respecting the ori… …code structure as much as possible) and a… short test program in front (generally a … loop)
    - Prefer… …henever there is a clearly predictable behaviour
- Purpo…
  - … discussion with compiler writers about their C++ …ompilers ("compiler critique")
    - Based on "real" codes with the possibility of quantifying the gain (in a given benchmark)

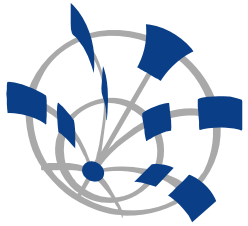
Purpose is NOT to identify snippets of code written in obscure ways (if they exist) !!

HEP = High Energy Physics



Potential pitfalls

- Several:
  - Code simplifications that lead to:
    - Optimization different from original code
    - Elimination of parts of the code
    - Single executable rather than sets of (shared) libraries
    - Different foot print (in cache, etc.)
- Nevertheless,
  - Inconveniences are outweighed by the advantages,
    - Especially the “instant reply” one (on any platform)

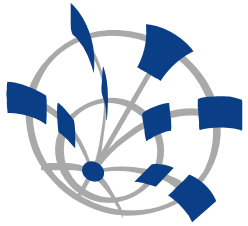


Extractions (so far)

- ROOT
  - TRandom3::Rndm
  - TRandom::Landau
  - TGeoCone::Contains
  - TGeoArb8::Contains
- CLHEP
  - RanluxEngine::Flat
  - HepRotation::rotateX
  - matrix::invertHaywood5
- GEANT4
  - G4Mag_UsualEqRhs::EvaluateRhsGivenB
  - G4Tubs::Inside
  - G4AffineTransform::InverseProduct

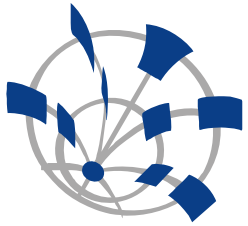
Obviously, more routines are needed for “full” coverage.
Potential candidates:
TBuffer::WriteFastBufferDouble
R_longest_match
TMatrixDSparse::AMultBt

But suggestions are welcome.



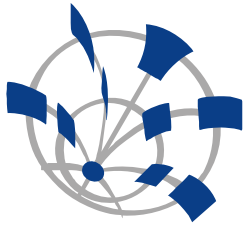
Methodology

- Three clear steps (which are easier with in-order processors than out-of-order ones):
  - Break sequence of C++ statements into machine instructions
    - Example: b = a + 1;
    - On RISC/EPIC:
      - Load
      - Add
      - Store
  - Estimate minimum number of cycles needed by an ideal compiler (or assembly program)
    - By exploiting parallel execution
      - Even (safe) speculative execution
  - Compare to the compiler at hand (using icc/gcc –S)



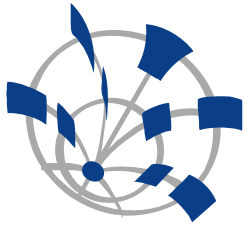
Where are the big payoffs?

- In my opinion (but not necessarily in priority order), compilers must achieve:
  - Comprehensive memory disambiguation
  - Correct amount of inlining
  - Aggressive extraction of parallelism
  - Fair amount of speculative execution
  - Full utilization of ALL architectural features
    - Execution units, parallel pipelines, registers, etc.



Now to the details

- Due to the short time for this talk only four cases will be reviewed:
  - G4AffineTransform::InverseProduct
  - TRandom::Landau
  - RanluxEngine::flat
  - TGeoCone::Contains
  - Compilers are:
    - gcc 3.2.3 and icc 9.0 (on 3.6 GHz Xeon/EM64T and 1.5 GHz Itanium-2)



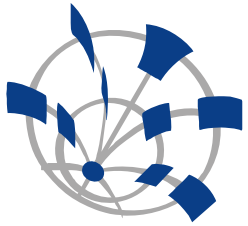
G4AffineTransform::InverseProduct

- The actual code:
  - Is a 3D rotation + translation
  - Very “clean” example, since the resource requirements are entirely clear:
    - 24 load’s
    - 45 fma’s
    - 12 stores

```
public:
  G4double rxx,rxy,rxz;
  G4double ryx,ryy,ryz;
  G4double rzx,rzy,rzz;
  G4double tx,ty,tz;
```

```
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; }
```

Best result: gcc on Xeon

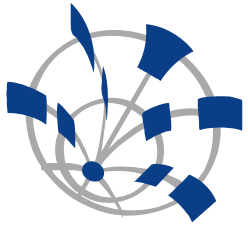


TGeoCone::Contains

- Simple routine, but several compiler challenges
  - Should *point[0]* and *point[1]* be loaded ahead of the first test (which only uses *point[2]*) ?
    - Should even the computation of *r2* start?
      - While we compute the outcome of the if statement
  - Should the two divisions be executed in parallel?

```
Bool_t TGeoCone::Contains(Double_t *point) const
{
// test if point is inside this cone
   if (TMath::Abs(point[2]) > fDz) return kFALSE;

   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;
   if ((r2<rl*rl) || (r2>rh*rh)) return kFALSE;
   return kTRUE;
}
```

Best result: gcc on Xeon



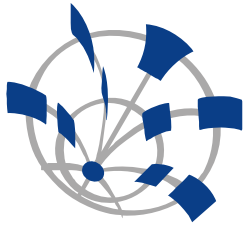
TRandom::Landau

- This is in the test suite for an entirely separate reason:

```
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)

   Double_t f[982] = {
      0       , 0       , 0       ,0        ,0        ,-2.244733,
     -2.204365,-2.168163,-2.135219,-2.104898,-2.076740,-2.050397,
   …… et cetera ……} ;
```

- Compilers can take strange paths for initializing this array
  - Copy bytes at a time onto the stack (Disastrous !!)
  - Copy doubles onto the stack (slightly better)
  - Use memcpy
  - Address the RODATA section directly (Best)

Best result: icc on Itanium

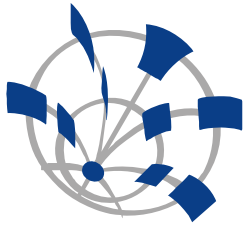


RanluxEngine::flat

- Again, one particular feature is the main interest
  - → See rhs
- Remember that this skip loop controls the luxury level:
  - The higher the luxury level, the more numbers we skip
- But the loop is difficult to optimize because of the loop-carried dependencies
  - Which, in turn, decides minimum loop latency

```
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;
      };
```

Best result: icc on Itanium

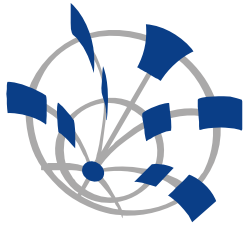


Overall comparison

- gcc 3.2.3 and icc 9.0 on a 3.6 GHz Xeon

On Itanium gcc is far behind, but it lacks specific Itanium-feature awareness

Xeon/64-bit 3.6 GHz O2 opt.	icc	gcc	%impr over gcc
TRandom3	7.46	7.06	94.6
TRandom	35.16	43.95	125.0
TGeoCone	10.29	9.24	89.8
TGeoArb8	5.5	5.64	102.5
RanluxEngine	16.8	15.2	90.5
HepRotation	9.65	17.2	178.2
MatrixInversion	22.1	26.0	117.6
G4Mag_EqRhs	9.1	8.97	98.6
G4Tubs	7.19	6.7	93.2
G4AffineTr.	6.9	6.71	97.2
GeoMean			**106.3**



Additional issues

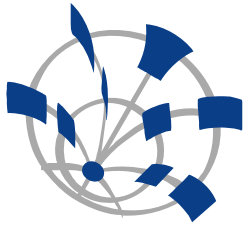
- A couple of other things:
  - Compile time of HepMatrix::invertHaywood[4,5,6] on Xeon:
    - gcc 3.2.3 (74 s)
    - icc 9.0 (1.7 s)
    - Fortunately, this improves with gcc 4.0.0 (5.1 s)
  - Student found that several routine ran more slowly with gcc4
    - Somewhat ironically, the MatrixInversion loses in execution what it gains in compilation.

Conclusions

- Optimization is still important:
  - LHC physicists have huge CPU demands
- In spite of the advent of multi-core pro…, Moore's law is under threat
  - And single processors will n… ach) faster
- When compil… able to generate faster code
  - Ever… nts
- …ealistic code snippets from HEP can help identify compiler weaknesses
  - Which, when corrected, could have a "global" impact

Please add realistic benchmarks to root/test if you want to help!

Backup



Plug for x86-64

(over x86-32)

- Compilers can produce better code (by default) because
  - A long series of architectural innovations are included by default
    - CMOV, FCMOV (Conditional moves)
    - MMX, etc.
    - SSE, SSE2 (for faster Floating-point calculations)
    - Extensions from 8 to 16 GP registers, 8 to 16 XMM registers

TGeoCone code

- gcc 3.2.3/ Xeon

```
.LFB1840:
        movsd   16(%rsi), %xmm2
        movsd   .LC2(%rip), %xmm0
        xorl    %eax, %eax
        movsd   8(%rdi), %xmm4
        andnpd  %xmm2, %xmm0
        ucomisd %xmm4, %xmm0
        ja      .L22
        movsd   (%rsi), %xmm5
        movsd   8(%rsi), %xmm0
        movapd  %xmm2, %xmm3
        movsd   32(%rdi), %xmm1
        addsd   %xmm4, %xmm3
        mulsd   %xmm0, %xmm0
        mulsd   %xmm5, %xmm5
        addsd   %xmm0, %xmm5
        movapd  %xmm4, %xmm0
        mulsd   %xmm3, %xmm1
        subsd   %xmm2, %xmm0
        mulsd   40(%rdi), %xmm3
        movapd  %xmm0, %xmm2
        movsd   16(%rdi), %xmm0
        mulsd   %xmm2, %xmm0
        mulsd   24(%rdi), %xmm2
        addsd   %xmm0, %xmm1
        movsd   .LC3(%rip), %xmm0
        addsd   %xmm2, %xmm3
        mulsd   %xmm0, %xmm1
        mulsd   %xmm0, %xmm3
        divsd   %xmm4, %xmm1
        divsd   %xmm4, %xmm3
        mulsd   %xmm1, %xmm1
        ucomisd %xmm5, %xmm1
        ja      .L28
        mulsd   %xmm3, %xmm3
        movl    $1, %eax
        ucomisd %xmm3, %xmm5
        jbe     .L22
.L28:
        xorl    %eax, %eax
.L22:
        ret
```