Software kernels

Sverre Jarp – CERN openlab

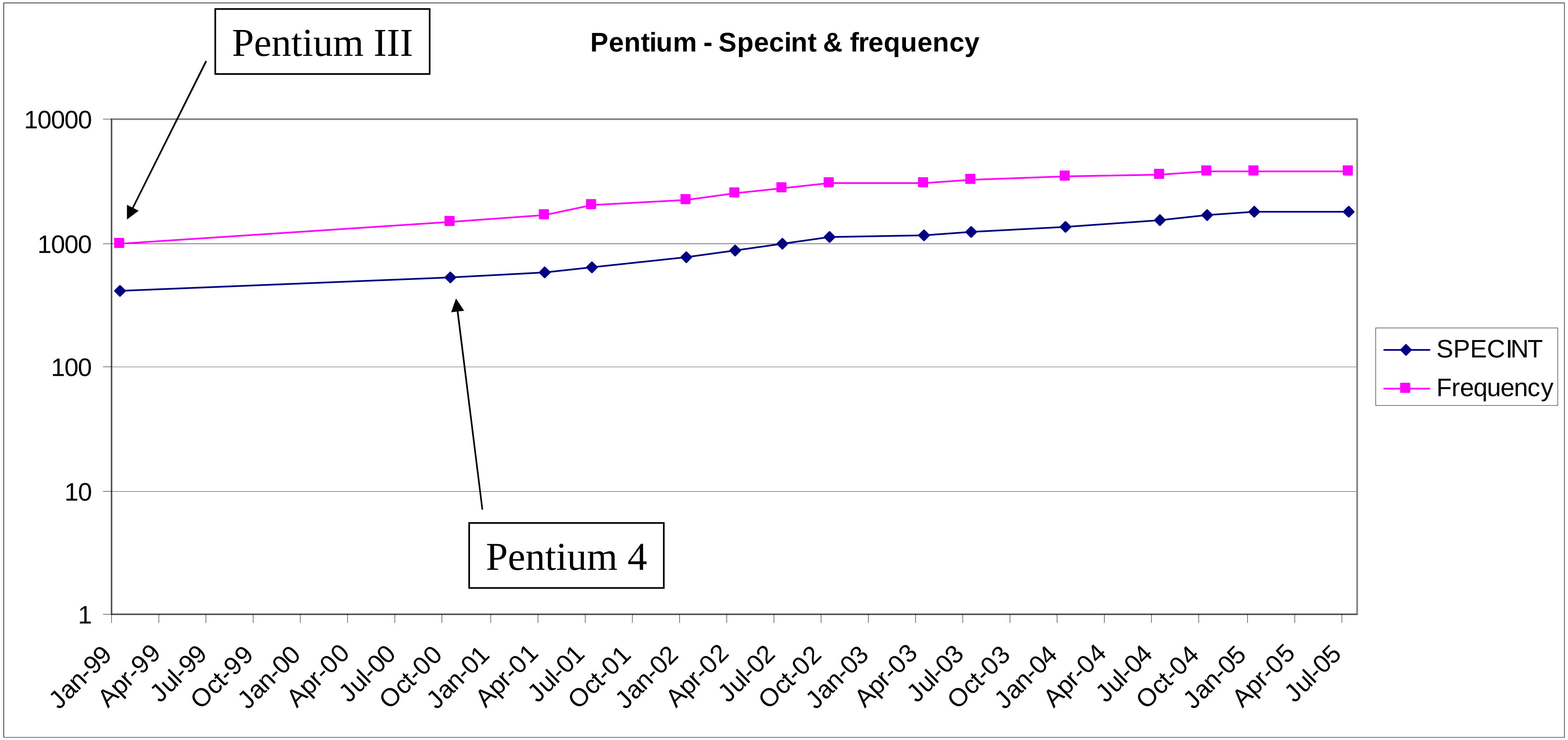
CHEP06 - 15 February 2006

Agenda

- Why look at performance?
- HEP programs and their execution profile
- Extraction of snippets
- Execution behaviour and comparisons
- Conclusion

Uni-processor performance

- **Practically flat since 2003:**



Rationale

- **Since frequency increases are “few and far between”**
  - Uni-processor performance improvement has to come from
    - Micro-architectural improvements
    - Compiler optimization improvements
- **Keep in mind:**
  - Multicore/many-core (per die) will continue to increase throughput
- **Moore’s law only promised:**
  - “transistor budget will double”

What is the issue?

```
Command: ./bench
Flat profile of CPU_CYCLES in bench-pid9051-cpu0.hist#0:
 Each histogram sample counts as 1.00034m seconds
% time     self     cumul     calls self/call  tot/call name
  5.04     4.66     4.66     46.9M     99.5n      131n TRandom::Gaus(double, double)
  4.68     4.33     8.99     63.7M     68.0n         - R__longest_match
  4.42     4.09    13.08         -         -         - _init<libCore.so>
  3.77     3.49    16.57     77.8M     44.9n     94.0n malloc
  3.30     3.05    19.62     20.2M      151n      151n TBuffer::WriteFastArray(int const*, int)
  3.24     3.00    22.62     77.6M     38.7n     50.7n __libc_free
  3.08     2.85    25.47     77.1M     37.0n     49.5n _int_malloc
  3.07     2.84    28.31     20.5M      138n      138n TBuffer::ReadFastArray(int*, int)
  2.92     2.70    31.01     6.82k      396u         - R__Deflate_fast
  2.79     2.58    33.60     7.20k      359u      514u R__Inflate_codes
  2.27     2.10    35.70      168M     12.5n     12.5n R__send_bits
  2.11     1.95    37.65     4.67M      418n      956n int
TStreamerInfo::ReadBuffer<char**>(TBuffer&, char** const&, int, int, int, int)
  1.98     1.83    39.48     30.2M     60.6n     60.6n TExMap::FindElement(unsigned long, long)
…………………………………..
```

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

Over time, more routines will be selected.

Potential candidates:
TBuffer::WriteFastBufferDouble
R_longest_match
TMatrixDSparse::AMultBt

Ongoing effort.

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 (when talking to compiler writers),
    - Especially the “instant reply” one (on any platform)

Now to the details

- **In this talk only half of the cases will be reviewed:**
  - G4AffineTransform::InverseProduct
  - TGeoCone::Contains
  - RanluxEngine::flat
  - Matrix::invertHaywood5
  - TRandom::Landau

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
  - Compiler must do full memory disambiguation

```
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; }
```

TGeoCone::Contains

- **Simple routine with a couple of 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?
    - When can the divisions be relaxed to multiplications with the reciprocal?

```
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;
}
```

RanluxEngine::flat

- **Again, one particular feature is the main interest**
  - → Loop carried dependencies

- **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 these 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;
      };
```

Matrix_Inversion

- **Depends on one inlining operation:**

```
Command: ./testMatrixInversion
Flat profile of CPU_CYCLES in testMatrixInver-pid32105-cpu0.hist#0:
 Each histogram sample counts as 1.00032m seconds
% time    self    cumul     calls self/call  tot/call name
 96.62    13.37    13.37     100M      134n      134n HepMatrix::invertHaywood5(int&)
  2.21     0.31    13.67        -         -         - main
  1.10     0.15    13.82        -         -         - ixgb_link_reset<kernel>
  0.02     0.00    13.83    16.6k      181n      181n _spin_unlock_irqrestore<kernel>
```

```
Command: ./testMatrixInversion
Flat profile of CPU_CYCLES in testMatrixInver-pid770-cpu0.hist#0:
 Each histogram sample counts as 1.00032m seconds
% time    self    cumul     calls self/call  tot/call name
 77.91   131.59   131.59    99.2M     1.33u     1.67u HepMatrix::invertHaywood5(int&)
 19.84    33.51   165.10    26.5G     1.26n     1.26n
                                   std::vector<double, std::allocator<double> >::operator[](unsigned long
  1.24     2.10   167.21        -         -         - ixgb_link_reset<kernel>
  0.98     1.65   168.86        -         -         - main
  0.01     0.02   168.87     226k     79.7n     79.7n _spin_unlock_irqrestore<kernel>
```

TRandom::Landau

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

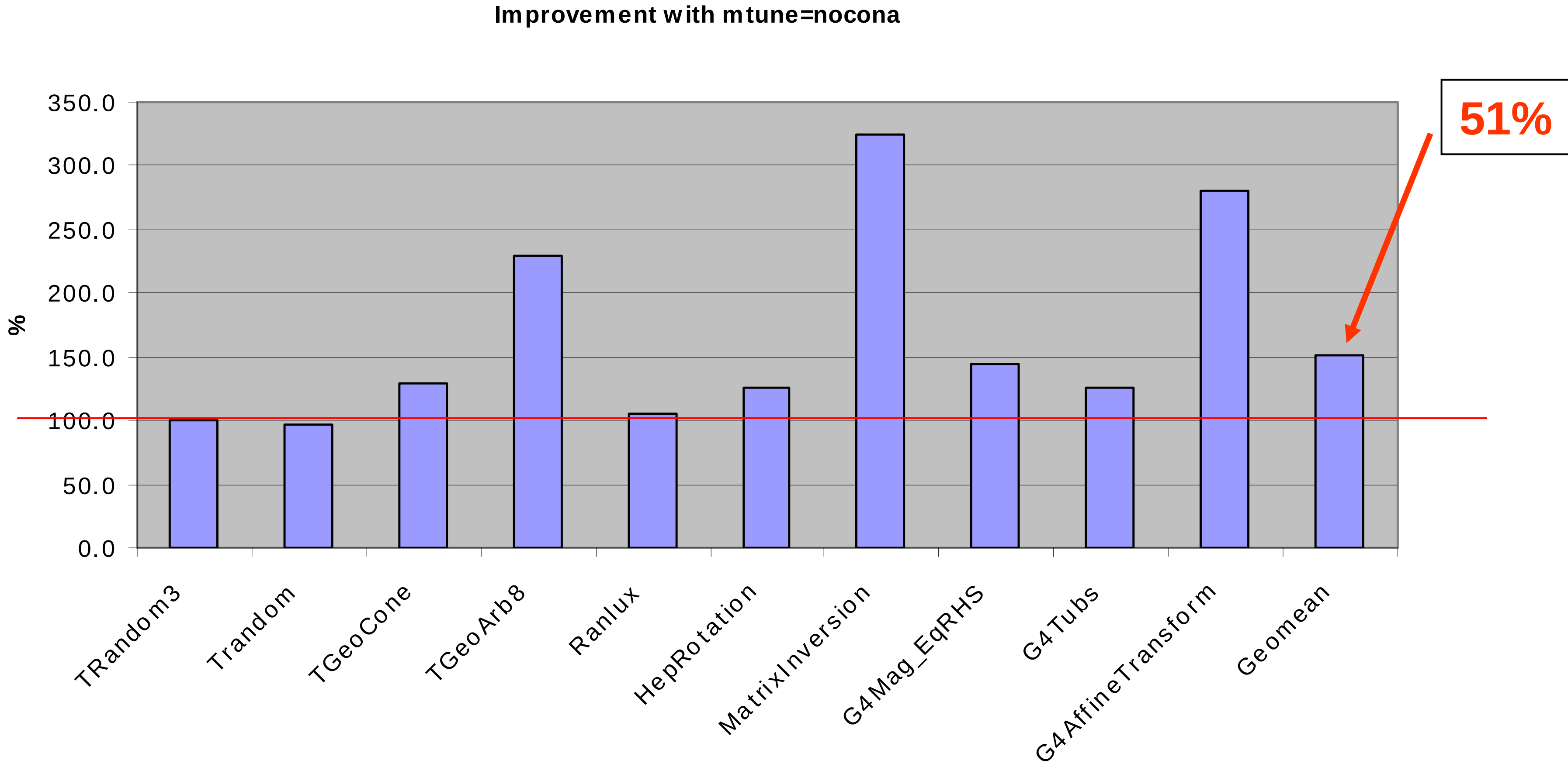
```
Double_t TRandom::Landau(Double_t mpv, Double_t sigma)
{
//  Generate a random number following a Landau distribution
//  with mpv(most probable value) and sigma

   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 (which is NOT declared *const* or *static*)**
  - Copy bytes at a time onto the stack (Disastrous !!)
  - Copy doubles onto the stack (slightly better)
  - Use memcpy
  - Verify that it can address the RODATA section directly (Best)

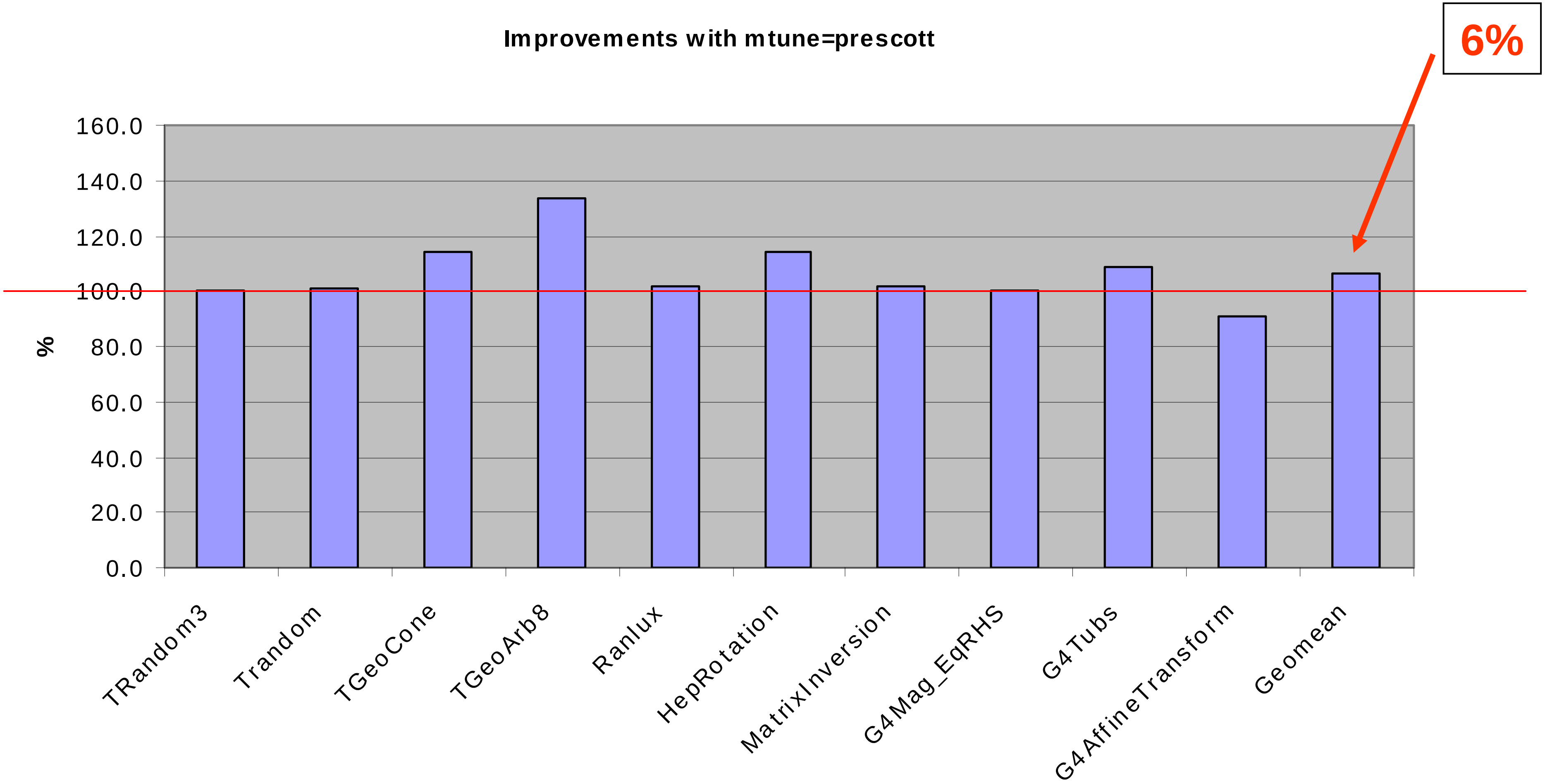
Used in comparisons (1)

- gcc 4.0.2 **(with O2) on a 3.6 GHz Xeon (64-bit mode)**
  - With/without mtune=nocona



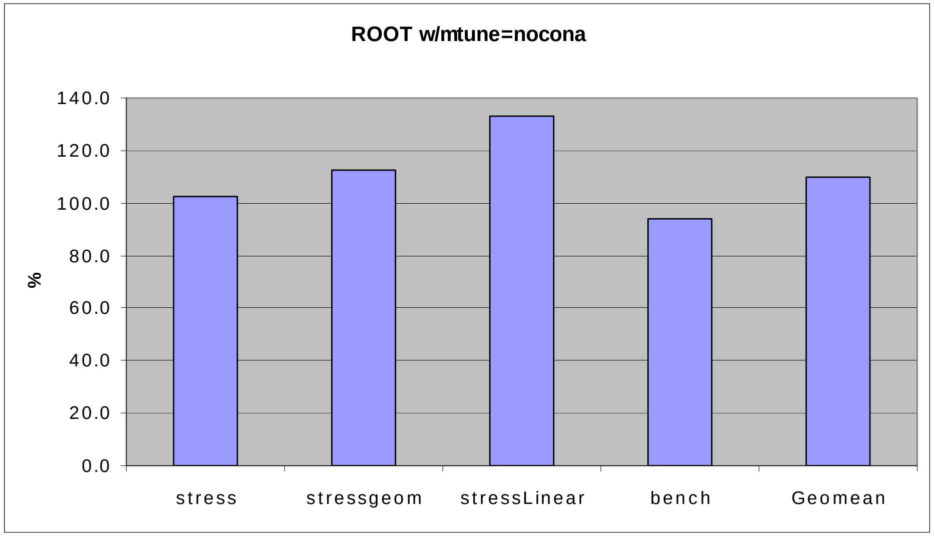
Used in comparisons (2)

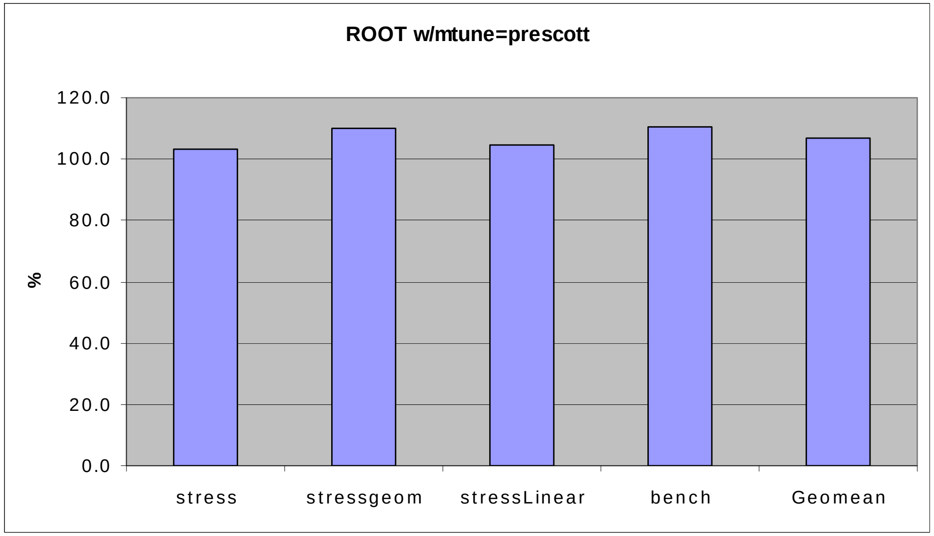
- gcc 4.0.2 (with O2) on a 2.4 GHz Xeon (32-bit mode)
  - With/without mtune=prescott



When moving to ROOT

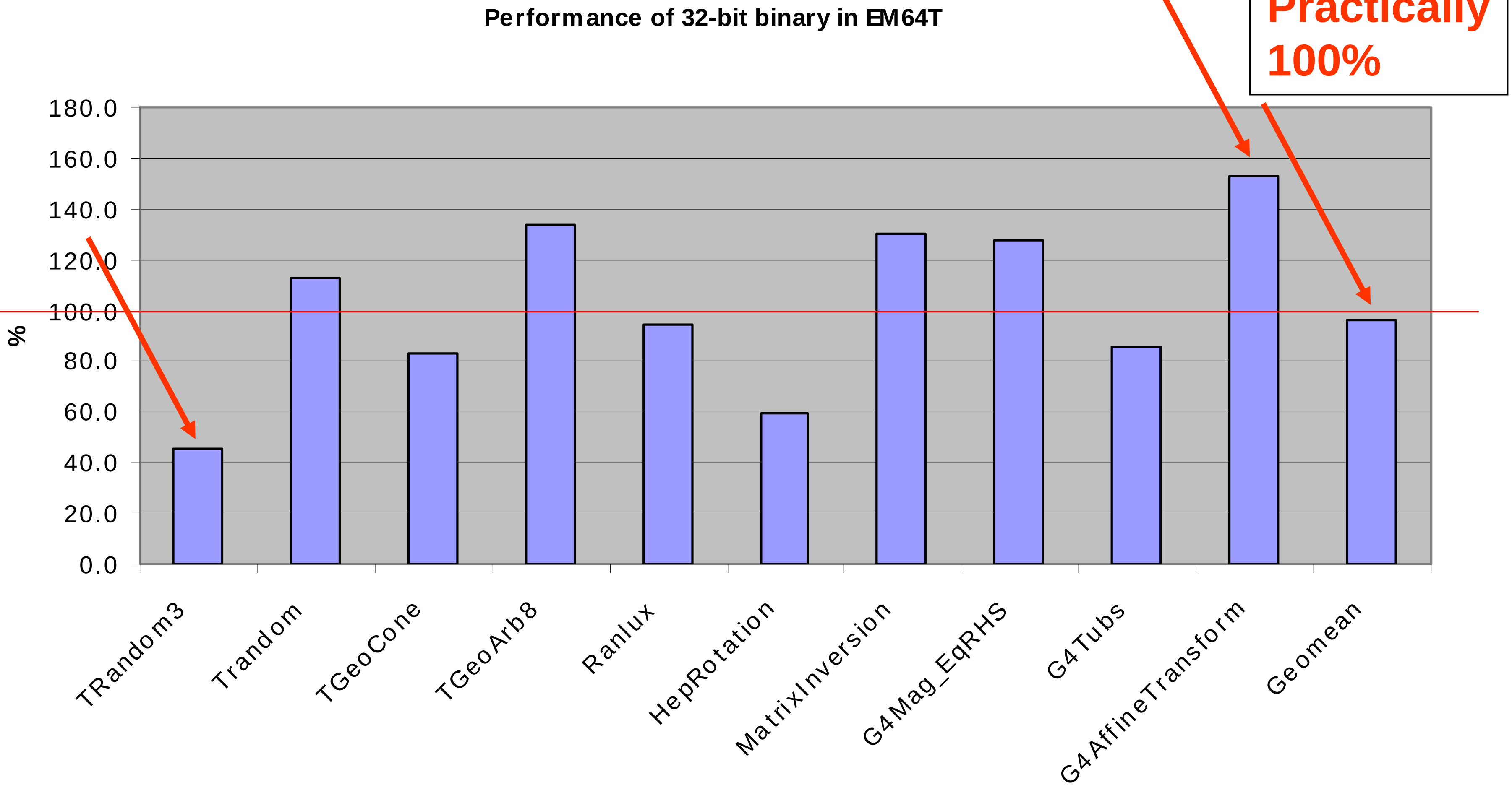
- **Snippets:**
  - Mtune=nocona (151%)
  - Mtune=prescott (106%)
- **Become:**
- **ROOT**
  - Mtune=nocona (110%)
  - Mtune=prescott (107%)





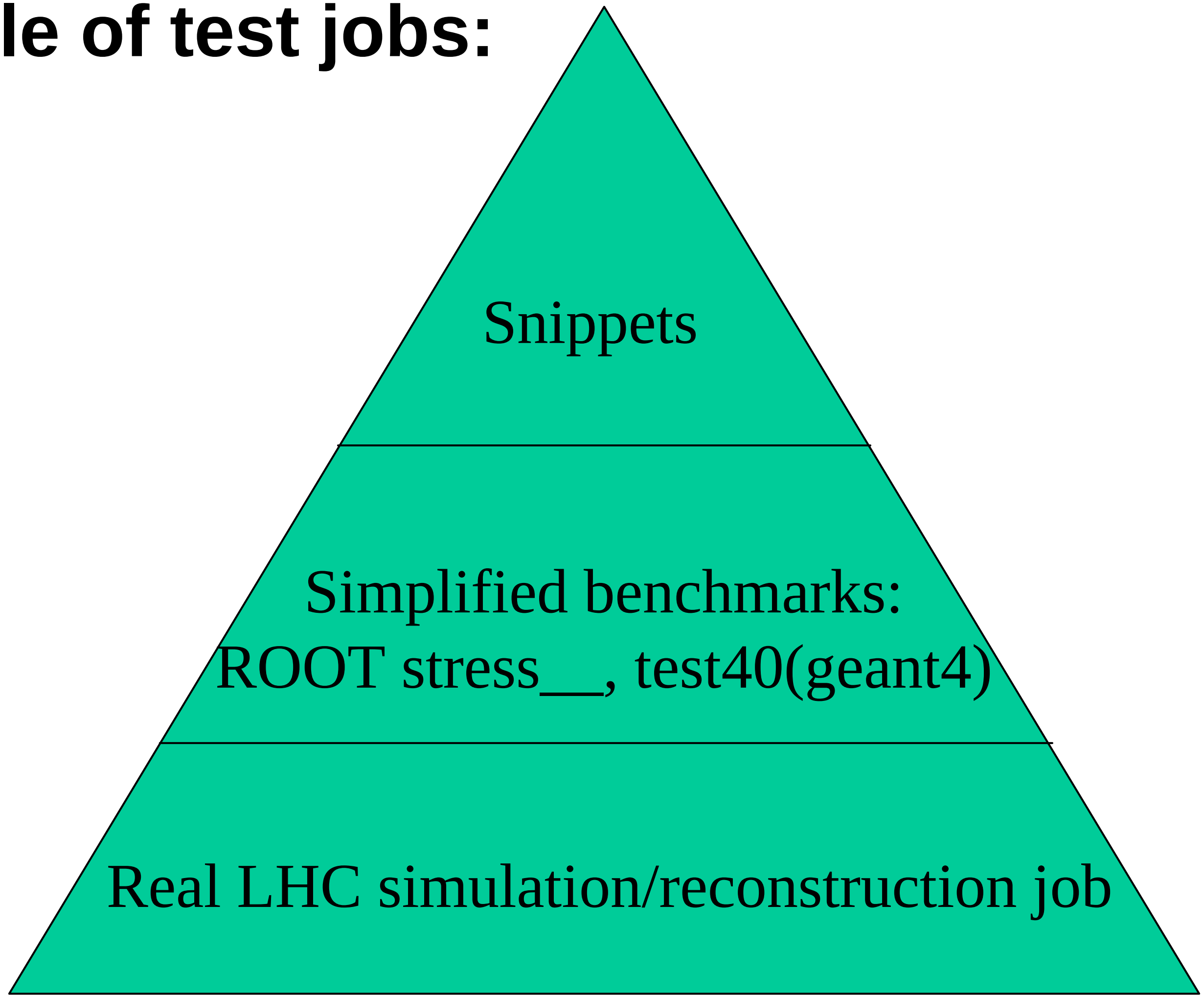
Other tidbits (1)

- **Moving 32-bit binaries to EM64T:**



The pyramid

- **Profile of test jobs:**



Snippets

Simplified benchmarks:
ROOT stress__, test40(geant4)

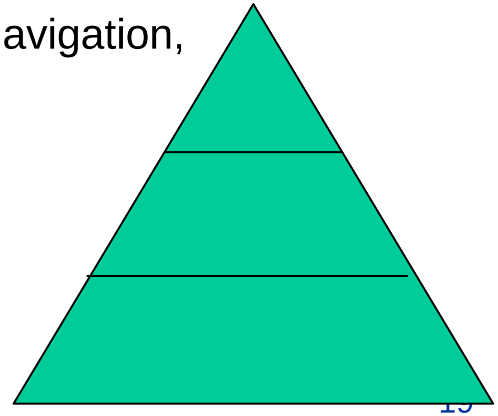
Real LHC simulation/reconstruction job

Conclusions

- **Benchmarking and optimization are still important:**
  - LHC physicists will have huge CPU demands

- **But, we have to tread carefully!**
  - “You only test what you REALLY test”

- **As we have seen:**
  - Snippets: Great for testing single compiler features
    - **Mandatory** in discussions with compiler writers
  - ROOTmarks (from *stress* testing)
    - Need to know our domain (file input/output, geometrical navigation, Linear Algebra, STL, etc.)
  - The full-blown LHC applications
    - Best – but extremely complex to port



Backup