Making a Data Product Thread-Safe

Christopher Jones
Brian Bockelman
Matevz Tadel

Thread Safe Products

All Event data products must be thread safe

Can be accessed by multiple modules simultaneously

Documentation on how to be thread safe is available

https://twiki.cern.ch/twiki/bin/view/CMSPublic/FWMultithreadedThreadSafeDataStructures

C++11 and thread safety

C++11 assumes talking to objects via const member functions is thread safe

Standard CMSSW Rules

const member functions must return the same value given the same arguments

i.e. caching is allowed but must be made explicitly thread safe

const functions must not return non-const pointers or references

would allow thread 1 to be changing internals of product while thread 2 is reading it

only const interactions are allowed

no 'const cast' of data products is allowed

implementation must not use non-const statics

thread 1 could be changing the static while thread 2 is reading it

CMS has a tool which finds such problems

based on clang static analyzer

http://cmssdt.cern.ch/SDT/scan-build/2013-03-05-1/

Choosing the Victim

Worked on reco::Particle

Caching strategy is same as many other classes

```
/// four-momentum Lorentz vector
float pt_, eta_, phi_, mass_;
...
/// internal cache for p4
mutable PolarLorentzVector p4Polar_;
/// internal cache for p4
mutable LorentzVector p4Cartesian_;
/// has cache been set?
mutable  bool cachePolarFixed_, cacheCartesianFixed_;
```

p4Polar_ and p4Cartesian_ are cached calculations of the momentum vector; cache* are set to true if the calculation has been performed. Not thread safe!

Caching increases size substantially

Caching takes 72 bytes (8[doubles]*4*2+2*1[bools]+6[padding])
Makes sizeof(Particle)=136 bytes

Updating the Cache

Request for data calls caching functions

```
const LorentzVector & p4() const
    { cacheCartesian(); return p4Cartesian_; }

const PolarLorentzVector & polarP4() const
    { cachePolar(); return p4Polar_; }
```

```
inline void cachePolar() const {
   if ( cachePolarFixed_ ) return;
   p4Polar_ = PolarLorentzVector( pt_, eta_, phi_, mass_ );
   cachePolarFixed_ = true;
}

inline void cacheCartesian() const {
  if ( cacheCartesianFixed_ ) return;
  cachePolar();
  p4Cartesian_ = p4Polar_;
  cacheCartesianFixed_ = true;
}
```

Thread Safety Issues

```
inline void cachePolar() const {
   if ( cachePolarFixed_ ) return;
   p4Polar_ = PolarLorentzVector( pt_, eta_, phi_, mass_ );
   cachePolarFixed_ = true;
}
```

Compiler can reorder operations

cachePolarFixed_ can be set before p4Polar_

CPU can reorder operations

Even if machine code has p4Polar_ set before cachePolarFixed_ CPU can reorder

Updates can appear to happen in different order across CPUs

If 2 different CPUs are calling cachePolar for the same object the CPU cache holding cachePolarFixed_ may be newer than p4Polar_

Can get partial results

p4Polar_ could be partway through being changed while someone is reading it

C++11 says you must synchronize all updates across threads

Paths to Thread Safety

Drop caching all together

- Could just create a temporary each time user requests
- Could initialize values in constructor
- Could get rid of slow interfaces entirely

Use mutex to synchronize

Use std::atomic to synchronize

Cost Of Not Caching

```
const LorentzVector & p4() const
    { cacheCartesian(); return p4Cartesian_; }

const PolarLorentzVector & polarP4() const
    { cachePolar(); return p4Polar_; }
```

Function	First Call	Later Calls
p4()	190ns	2.6ns
polarP4()	5ns	2.6ns

Optimizing polarP4

1. Make a new copy each call : 5.5 ns

The ROOT::Math::PtEtaPhiM4D<double> constructor is heavy weight

normalizes value of Phi
checks for negative value of mass and treats it as it if were energy instead

2. Bypassed constructor call and used floats, return ref: 2.2 ns

Made pt_,eta_,phi_,mass_ already in Particle align to storage of polarP4
Did 'cast magic' to return ref to internal data
NOTE: Use of ROOT object as member data would increase I/O storage cost

Size decreased to 104 bytes

Caching p4

Changed p4 to be floats instead of doubles

sizeof(Particle) == 80 bytes

Bypassed constructor call ala polarP4

Directly cached px,py,pz,energy

Same speed: 2.6ns

p4 via Mutex

```
mutable bool cacheCartesianFixed_;
mutable std::mutex cache_mutex;
```

```
inline void cacheCartesian() const {
  std::lock_guard<std::mutex> lock(cache_mutex);
  if ( cacheCartesianFixed_ ) return;
  const SmallLorentzVector tmpP4( polarP4() );
  px_ = tmpP4.Px();
  py_ = tmpP4.Py();
  pz_ = tmpP4.Pz();
  energy_ = tmpP4.E();
  cacheCartesianFixed_ = true;
}
```

Slowed way down: 20ns

Increased memory size by 48 bytes

Atomic with Wait

```
mutable std::atomic<char> cacheCartesianFixed_;
```

```
inline void cacheCartesian() const {
  char cache_val = cacheCartesianFixed_.load();
  if (likely(cache_val ==kCached)) return;
  cache_val = kInit;
  if (cacheCartesianFixed_.compare_exchange_strong(cache_val, kUpdating)){
    const SmallLorentzVector tmpP4( polarP4() );
    px_ = tmpP4.Px(); py_ = tmpP4.Py();
    pz_ = tmpP4.Pz(); energy_ = tmpP4.E();
    cacheCartesianFixed_.store(kCached);
  } else if (cache_val == kCached){
    return;
  } else {
    while(cacheCartesianFixed_.load() != kCached) {}
  }
}
```

Checks if already cached and if so returns [likely]

likely() improved solution performance by about 50%. Better assembly code.

Then tries to be the one to update the value

If not updating then waits for other thread to finish update

Good speed: 2.2ns

Atomic with Copy

```
mutable std::atomic<char> cacheCartesianFixed ;
```

```
SmallLorentzVector p4() const {
  char cache_val = cacheCartesianFixed_.load();
  if (likely(cache_val == kCached)) return p4Cached_;
  cache_val = kInit;
  if (likely(cacheCartesianFixed_.compare_exchange_strong(cache_val, kUpdating))) {
    p4Cached_ = polarP4();
    __sync_synchronize();
    cacheCartesianFixed_.store(kCached);
    return p4Cached_;
  }
  return SmallLorentzVector( polarP4() );
}
```

Checks if already cached and if so return p4Cached_ [likely]

Try to be the one to update the value

If not, calculate a temporary and return it

Good speed: 2.3ns

Predicable behavior if more than one thread attempts update

Conclusion

Possible to maintain performance while making thread safe

Easiest solution is to drop cache all together

Need to redo performance measurements to justify caching

May need to explore multiple thread safe caching strategies

Different classes may need different approaches

Users need thread safe caching patterns they can follow