Memory Leakage Monitoring

S. Y. Jun (Fermilab), G. Cosmo (CERN), A. Dotti (SLAC)

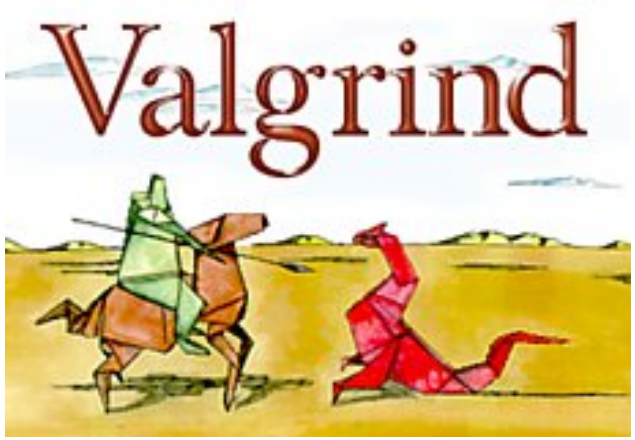
20th Geant4 Collaboration Meeting at Fermilab
Sept. 28 - Oct. 2, 2015

Introduction

- C++ memory use and management
  - Performance (memory churn, locality, contention)
  - Correctness
  - No (implicit) garbage collection – be clear on ownership
- Do small memory leaks matter?
  - The system cleans up unreleased memory (by the OOM killer) if all of available memory (main + swap) is consumed - a potential danger of application slowness or system locking up
  - Indication of a poor design for ownership or lifetime of objects
- Leaks from Geant4? - relatively clean. Two distinct types:
  - Memory allocated at initialization, but not explicitly released at the end of program (the majority of the cases, less critical)
  - Memory allocated within the event loop, but not freed (the most critical and relevant for production runs in the experiments)

Problem Statement and Tools

- Problem Statement:
  - Reduce existing memory leaks
  - Monitor newly introduced leaks
- Tools
  - lgprof (a low-overhead memory profiler)
  - Valgrind (a great tool for memcheck, but too slow)
  - Coverity (a static code analysis)
  - a custom monitoring tool (under developing)



coverity®

IgProf

- A primary memory profiling tool for the Geant4 computing performance task
- A great tool for tracing memory churns (MEM TOTAL), but not sensitive to small amount of memory leaks (difference in MEM LIVE between 1st and Nth event) out of Geant4 source codes

Geant4.10.1.r08 SimplifiedCalo B=4.0T https://oink.fnal.gov/perfanalysis/g4p/

Sample	Physics List	Energy	MEM_LIVE	MEM_MAX	MEM_TOTAL
Higgs->ZZ	FTFP_BERT	14 TeV	1 Diff(51-1) 51 End of Run	1 Diff(51-1) 51 End of Run	1 Diff(51-1) 51 End of Run
Electrons	FTFP_BERT	5 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
		50 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
	FTFP_BERT	5 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run
		50 GeV	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run	1 Diff(1001-1) 1001 End of Run

Champaign I: Valgrind

- The test and QA (G. Cosmo) runs Valgrind tests
  - Each official release
  - 19 representative tests
  - Check critical leaks mainly within the event loop
  - Execute two independent runs with different statistics and compare outputs
- Geant4 code being released is relatively clean

```
01  B2b
02  B4b
03  test02  (em)  # FTFP_BERT Physics Lists
04  test02  (had) #  FTFP_BERT Physics Lists
05  test11        #  Neutron transport.
06  test12        #  FTF String + precompound.
07  test13        #  QGSM, Dual Parton+precompound.
08  test14  (low) #  LowEnergy em (photons and e-).
09          (pen) #  penelope
10          (pol) #  polarised
11  test16        #  n and p Cross-Sections
12  test17        #  LowEnergy e.m. (p, anti-p, ions).
13  test18        #  Radioactive decay.
14  test24        #  Binary cascade hadronic model.
15  test25        #  Classical cascade hadronic model.
16  test27        #  Binary cascade for light ions
17  test28        #  Hadronic abrasion/em-dissociation.
18  test34        #  Shower parameterisation (GFLASH)
19  test60        #  Geant4-DNA processes.
```

Output: /afs/cern.ch/sw/geant4/dev/QA_tools/Valgrind/logs/

Valgrind Memcheck : 9 Different Cases

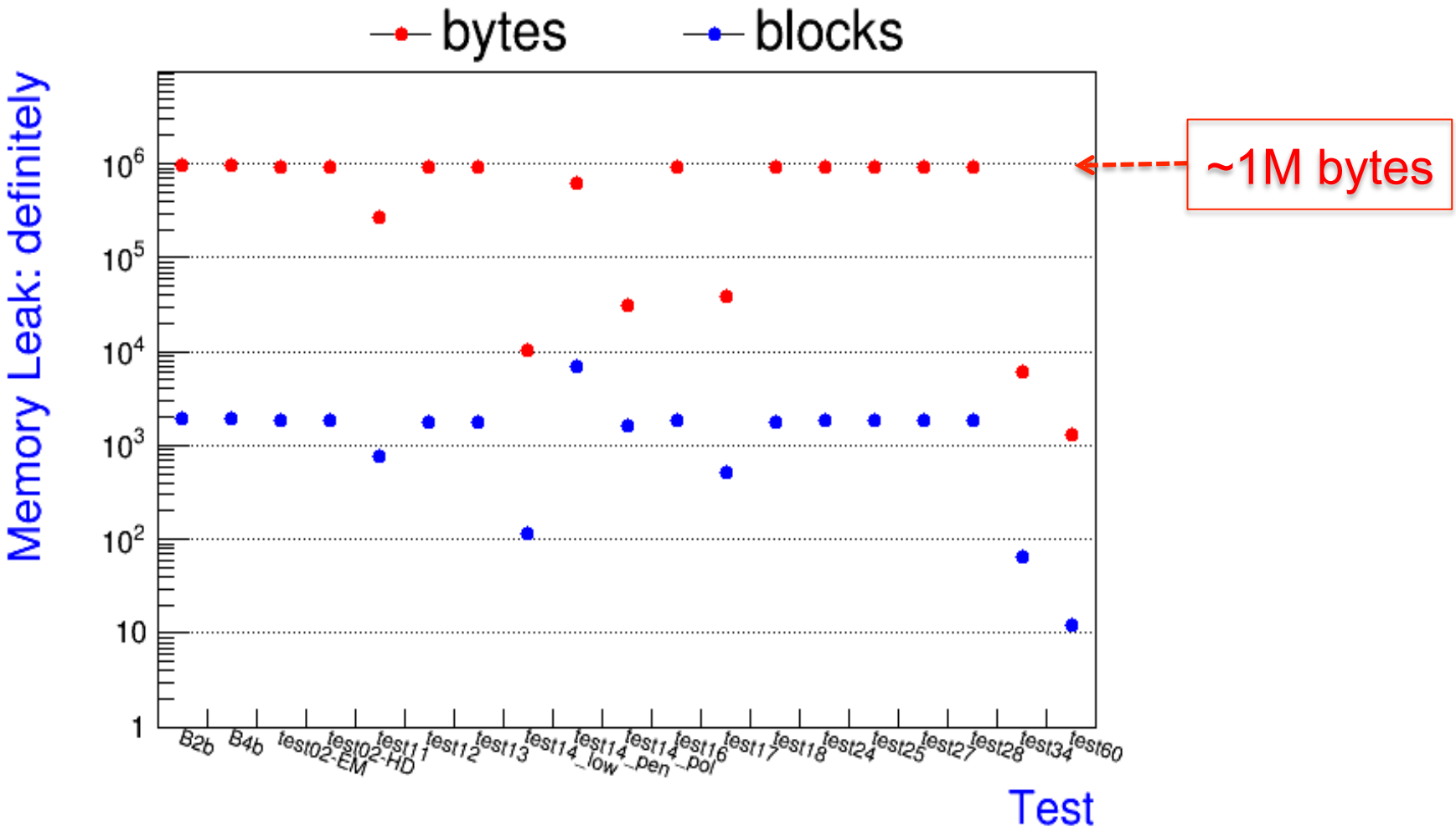
- Directly/Indirectly reachable (DR,IR): 1, 2 - arguably not a problem
- Definitely lost (DL): 3 – should be fixed
- Indirectly lost (IL): 4, 9 – O.K. if DL is fixed (ex: a binary tree)
- Possible lost: 5-8 – make sure that inter-pointers exist

	Pointer chain	AAA Leak Case	BBB Leak Case
(1)	RRR ----------> BBB		DR
(2)	RRR ---> AAA ---> BBB	DR	IR
(3)	RRR BBB		DL
(4)	RRR AAA ---> BBB	DL	IL
(5)	RRR ------?-----> BBB		(y)DR, (n)DL
(6)	RRR ---> AAA -?-> BBB	DR	(y)IR, (n)DL
(7)	RRR -?-> AAA ---> BBB	(y)DR, (n)DL	(y)IR, (n)IL
(8)	RRR -?-> AAA -?-> BBB	(y)DR, (n)DL	(y,y)IR, (n,y)IL
(9)	RRR AAA -?-> BBB	DL	(y)IL, (n)DL

Pointer chain legend:
- RRR: a root set node or DR block
- AAA, BBB: heap blocks
- --->: a start-pointer
- -?->: an interior-pointer

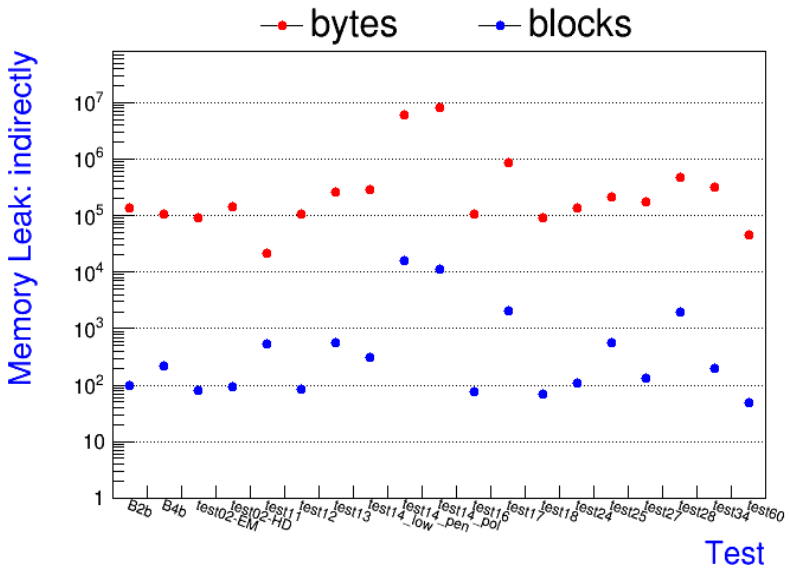
Valgrind Summary (I) : Geant4 10.2.beta

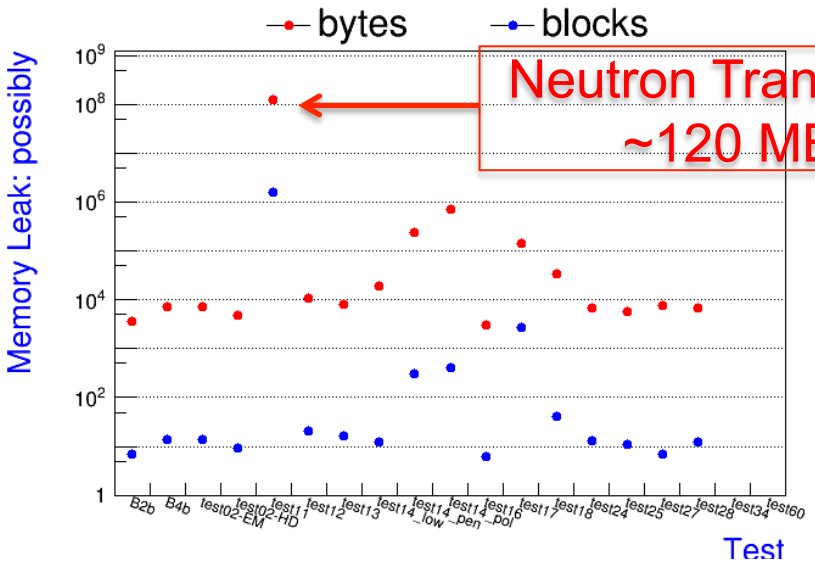
- Definitely Lost: no pointer to the block can be found (lost the pointer at the earlier point)

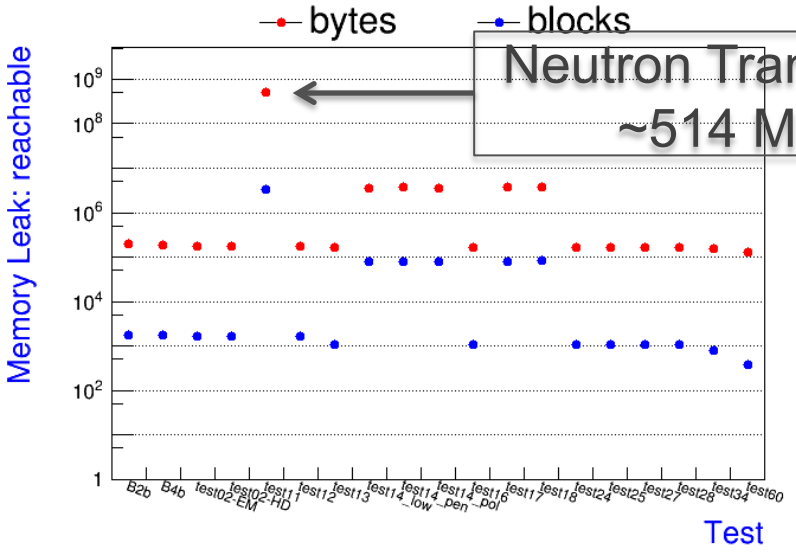


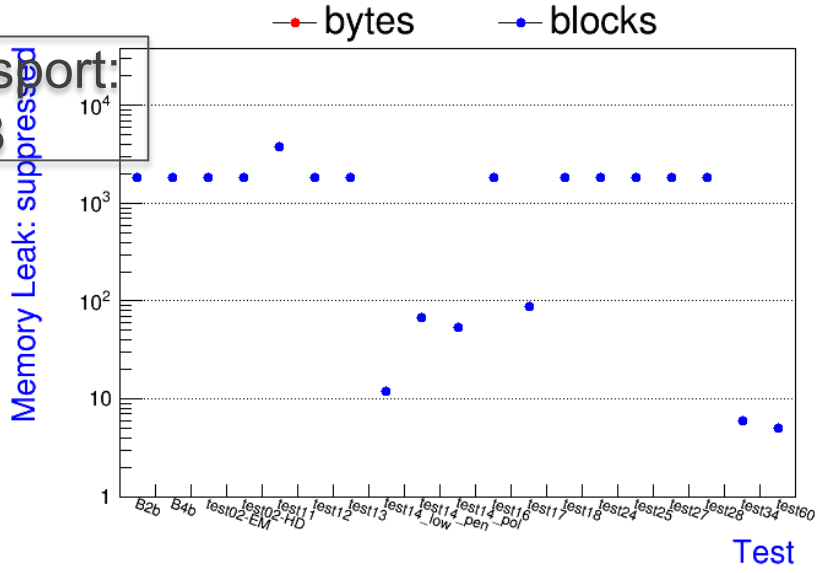
Valgrind Summary (2): Geant4 10.2.beta

- Indirectly lost, Possibly lost, Still Reachable, Suppressed



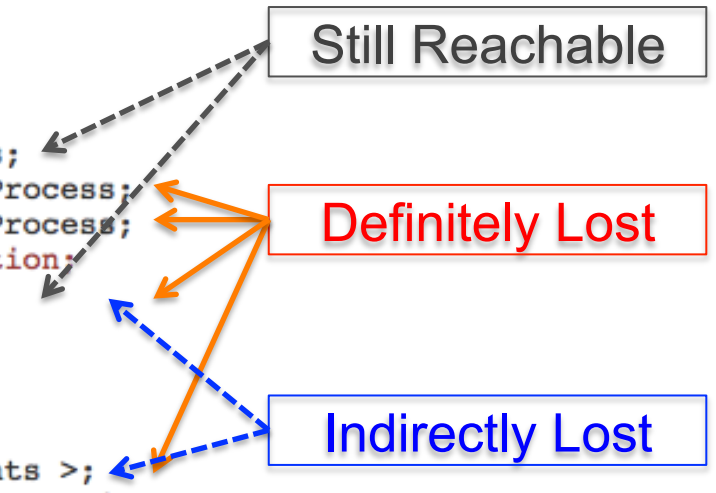






Valgrind Report: Example I

```
63 G4BertiniElectroNuclearBuilder::~G4BertiniElectroNuclearBuilder()
64 {
65   if(wasActivated) {
66     delete theFragmentation;
67     delete theStringDecay;
68     //delete theStringModel;
69     //delete thePhotoNuclearProcess;
70     //delete theElectronNuclearProcess;
71     //delete thePositronNuclearProcess;
72     //delete theElectroReaction;
73     //delete theGammaReaction;
74     //delete theModel;
75     //delete theCascade;
76   }
77 }
78
79 void G4BertiniElectroNuclearBuilder::Build()
80 {
81   if(wasActivated) return;
82   wasActivated=true;
83
84   thePhotoNuclearProcess = new G4PhotoNuclearProcess;
85   theElectronNuclearProcess = new G4ElectronNuclearProcess;
86   thePositronNuclearProcess = new G4PositronNuclearProcess;
87   //  theElectroReaction = new G4ElectroNuclearReaction;
88   theElectroReaction = new G4ElectroVDNuclearModel;
89   theGammaReaction = new G4CascadeInterface;
90
91   theModel = new G4TheoFSGenerator;
92
93   theStringModel = new G4QGSModel< G4GammaParticipants >;
```



Valgrind Report: Example 2

- One of most frequent (repetitive) definitely lost cases
  - The object (G4AngularDistribution) is properly allocated and deallocated, but lost inside containers in the call chain

```
520 bytes in 1 blocks are definitely lost in loss record 1,066 of 2,910
   at 0x4006578: operator new(unsigned int) (vg_replace_malloc.c:318)
   by 0x84C7FD3: G4VScatteringCollision::G4VScatteringCollision() (G4VScatteringCollision.cc:49)
   by 0x84C28AD: G4ConcreteNNTwoBodyResonance::G4ConcreteNNTwoBodyResonance(void*, void*, void*,
   by 0x84C211F: G4ConcreteNNToNDelta::G4ConcreteNNToNDelta(G4ParticleDefinition const*, G4Parti
   by 0x84BA2DE: operator()<INT4<G4ConcreteNNToNDelta, 2112, 2212, 2112, 2214> > (G4CollisionCom
   by 0x84BA2DE: call (G4Pair.hh:157)
   by 0x84BA2DE: Apply<G4CollisionComposite, G4CollisionComposite::Resolve> (G4Pair.hh:174)
   by 0x84BA2DE: Apply<G4CollisionComposite, G4CollisionComposite::Resolve> (G4Pair.hh:176)
   by 0x84BA2DE: Apply<G4CollisionComposite, G4CollisionComposite::Resolve> (G4Pair.hh:176)
   by 0x84BA2DE: Apply<G4CollisionComposite, G4CollisionComposite::Resolve> (G4Pair.hh:176)
   by 0x84BA2DE: Make (G4GeneralNNCollision.hh:64)
   by 0x84BA2DE: G4CollisionNNToNDelta::G4CollisionNNToNDelta() (G4CollisionNNToNDelta.cc:39)
   by 0x84AE4DD: call (G4Pair.hh:155)
   by 0x84AE4DD: Apply<G4CollisionNN, G4CollisionComposite::Register> (G4Pair.hh:174)
   by 0x84AE4DD: Apply<G4CollisionNN, G4CollisionComposite::Register> (G4Pair.hh:176)
   by 0x84AE4DD: Apply<G4CollisionNN, G4CollisionComposite::Register> (G4Pair.hh:176)
   by 0x84AE4DD: G4CollisionNN::G4CollisionNN() (G4CollisionNN.cc:60)
   by 0x84A9136: call (G4Pair.hh:155)
   by 0x84A9136: Apply<std::vector<G4VCollision*>, G4Scatterer::Register> (G4Pair.hh:174)
   by 0x84A9136: G4Scatterer::G4Scatterer() (G4Scatterer.cc:69)
   by 0x832223B: G4BinaryCascade::G4BinaryCascade(G4VPreCompoundModel*) (G4BinaryCascade.cc:128)
   by 0x832D38D: G4BinaryLightIonReaction::G4BinaryLightIonReaction(G4VPreCompoundModel*) (G4Bin
   by 0x806C5A6: G4IonPhysics::ConstructProcess() (G4IonPhysics.cc:132)
```

Valgrind Report: Example 3

- The biggest single definitely lost

```
85,864 (1,584 direct, 84,280 indirect) bytes in 18 blocks are definitely lost in loss record 2,910
   at 0x4006578: operator new(unsigned int) (vg_replace_malloc.c:318)
   by 0x871D109: G4VRangeToEnergyConverter::BuildLossTable() (G4VRangeToEnergyConverter.cc:317)
   by 0x871D540: G4VRangeToEnergyConverter::Convert(double, G4Material const*) (G4VRangeToEnergyCo
   by 0x871A436: G4ProductionCutsTable::UpdateCoupleTable(G4VPhysicalVolume*) (G4ProductionCutsTab
   by 0x82E12D7: G4RunManagerKernel::UpdateRegion() (G4RunManagerKernel.cc:691)
   by 0x82E2509: G4RunManagerKernel::RunInitialization(bool) (G4RunManagerKernel.cc:608)
```

```
for (size_t j=0; j<size_t(NumberOfElements); j++){
  G4double Value;
  G4LossVector* aVector= 0;
  aVector= new G4LossVector(LowestEnergy, MaxEnerg
  for (size_t i=0; i<=size_t(TotBin); i++) {
    Value = ComputeLoss(   (*G4Element::GetElementT
        aVector->Energy(i)
        );
    aVector->PutValue(i,Value);
  }
  theLossTable->insert(aVector);
```

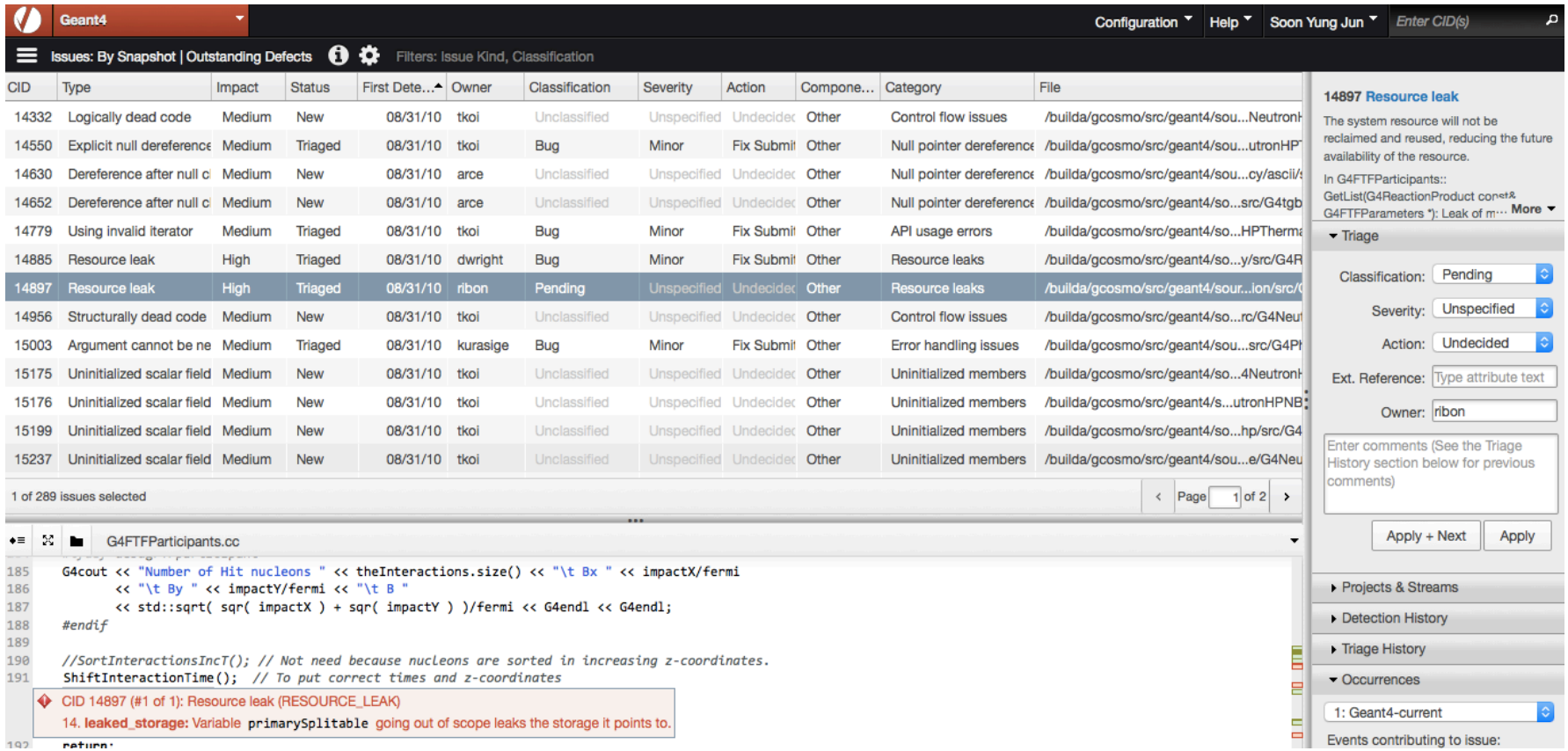
inappropriate deallocation!

```
G4VRangeToEnergyConverter::~G4VRangeToEnergyConverter(
{
  // Reset();
  // Comment out Reset() for MT application

  // delete loss table without deleteing vectors
  if (theLossTable) {
    delete theLossTable;
  }
  theLossTable=0;
```

Champaign 2: Coverity

- Static analysis (http://coverity.cern.ch/): Geant4: 289 issues
  - Resource leaks (39), Memory corruptions/illegal access (1/8)
  - Null pointer dereference (31), Uninitialized members (51), control flow issues (19), incorrect expressions (13), etc.



Coverity Category: Resource Leaks

- Two types of resource leaks under the Geant4 project
  - *new* on a data-member and does not free it. If data-type is
    - private: simply a bug (easy to fix)
    - protected or public: it could be trickier to solve in case the derived class or the user is actually deleting it explicitly
  - a *new* of an object in a method and no clear ownership
    - 14897 Resource leak: who owns primarySplitable?

```
G4InteractionContent::G4InteractionContent(G4VSplitableHadron *aPrimaryParticipant)
{
   theProjectile = aPrimaryParticipant;  // theProjectile is a private member of *this
}

void G4FTFParticipants::GetList( const G4ReactionProduct& thePrimary, …)
{
   G4VSplitableHadron* primarySplitable = new G4DiffractiveSplitableHadron( thePrimary );
        ……
   G4InteractionContent* aInteraction = new G4InteractionContent( primarySplitable );
}
```

Case Study (Exemplified by A. Dotti)

```
class G4Something;
class G4Class {
  G4Something* pointer;
   ~G4Class() { /*?? should I delete pointer??*/ }
   void set( G4Something* p) { pointer = p;}
   G4Something* get() const { return pointer; }
};

//Usage:
G4OtherClass::SomeMethod()
{
  G4Something* smt = new G4Something;
  G4Class* cls = new G4Class();
  cls->set( smt );
  //Who owns smt? Who should delete it?
}
```

Clear Object Ownership

- Use std::unique_ptr and move – make ownership explicitly

```
class G4Something;
class G4Class
{
   std::unique_ptr<G4Something> pointer;

   ~G4Class() {/* Do NOT do anything w/ pointer */}
   void set(std::unique_ptr<G4Something>&& p){pointer std::move(p);}

   const G4Something& get_const() const { return *pointer.get(); }
   std::unique_ptr<G4Something> get() { return std::move(pointer); }
};
//Usage:
  std::unique_ptr<G4Something> smt(new G4Something);
  G4Class* cls = new G4Class();

  //set(smt) will not compile, make explicit ownership
  cls->set(std::move(smt));

  //Cannot use anymore smt, not valid. Need to use it?
  const G4Something& smtref = cls->get_const();

  //Want ownership back or pass it to someone else:
  std::unique_ptr<G4Something> smt_own = cls->get();
```

Note (by Andrea)

- We need discussion how to use C++11 features of object ownership and a general agreement on it
- Remarks:
  - No memory overhead in simple use cases [1]:
    - sizeof(std::unique_ptr<double>) == sizeof(double*)
  - Overhead of the explicit ownership
    - void foo(std::unique_ptr<double>) requires one more assembler instruction than void foo(double*) - additional pointer dereference (negligible as long as foo uses the pointer in non trivial ways)[2]
  - rvalue references (i.e. double&&) are very tricky! We need to better understand them before start using them
- Attached test code to see it in action (unique_ptr.cc)

[1] http://stackoverflow.com/questions/13460395/how-can-stdunique-ptr-have-no-size-overhead
[2]: http://www.drdobbs.com/cpp/c11-uniqueptr/240002708

Campaign 3: A custom memory leak monitor

- Check unreleased memory at the exit of an application
  - Light-weighted and easy to analyze output (efficiency)
  - Complimentary to Valgrind (correctness)
- Push/pop memory allocation–deallocation (pointers on heap) during an application is running and dump undeleted pointers at the end program
- Principle is based on nvwa (a cross-platform memory leak detector): http://wyw.dcweb.cn/leakage.htm
- For Geant4 applications,
  - Build the static Geant4 library
  - Preload the library with custom global new/delete operators
  - Print file names and line numbers of unreleased memory addresses at the end of Geant4 examples or tests

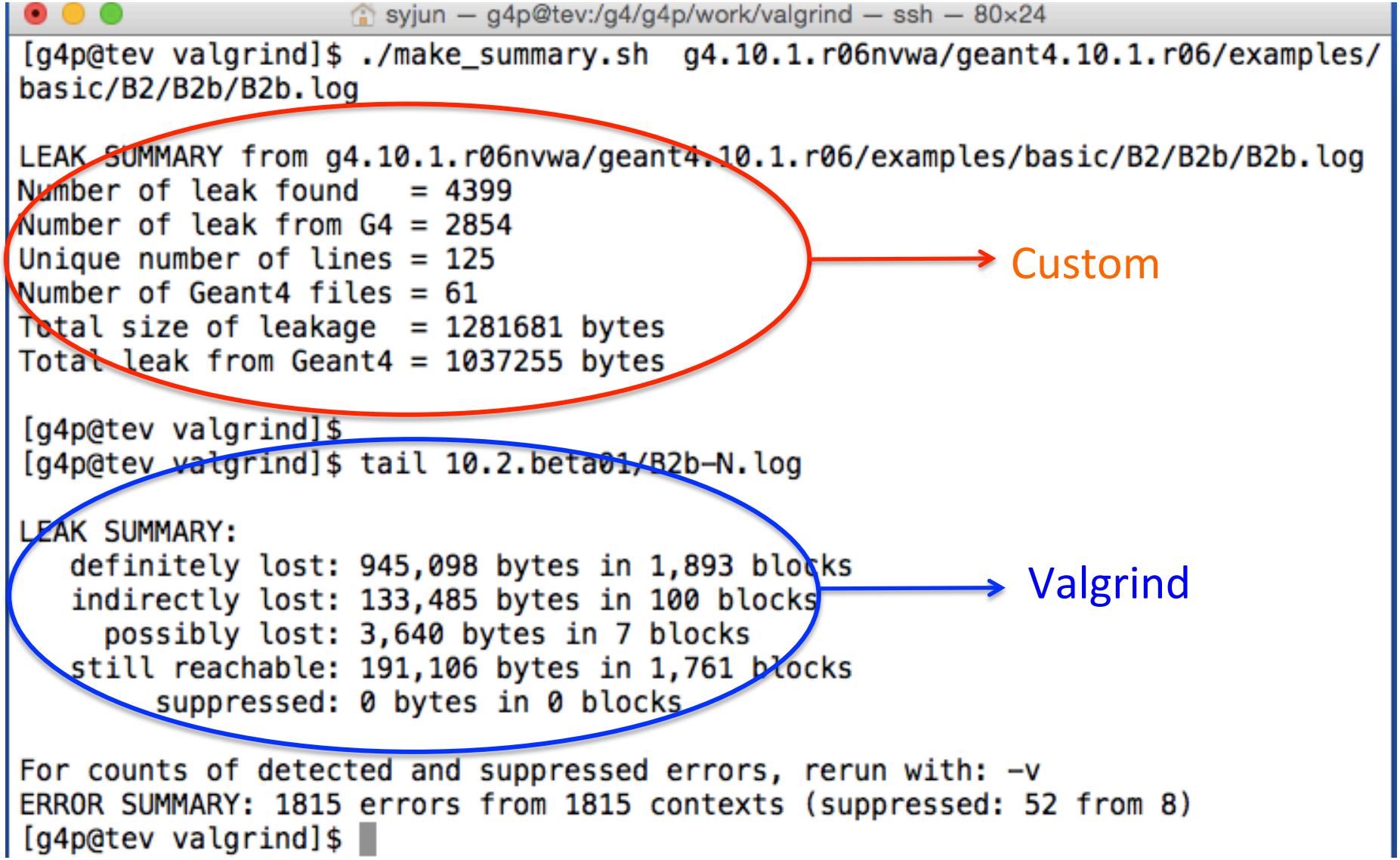
A custom memory leak monitor: Implementation Details

- Override new and delete (new[] and delete[]) with custom operators by adding/removing the address of the caller

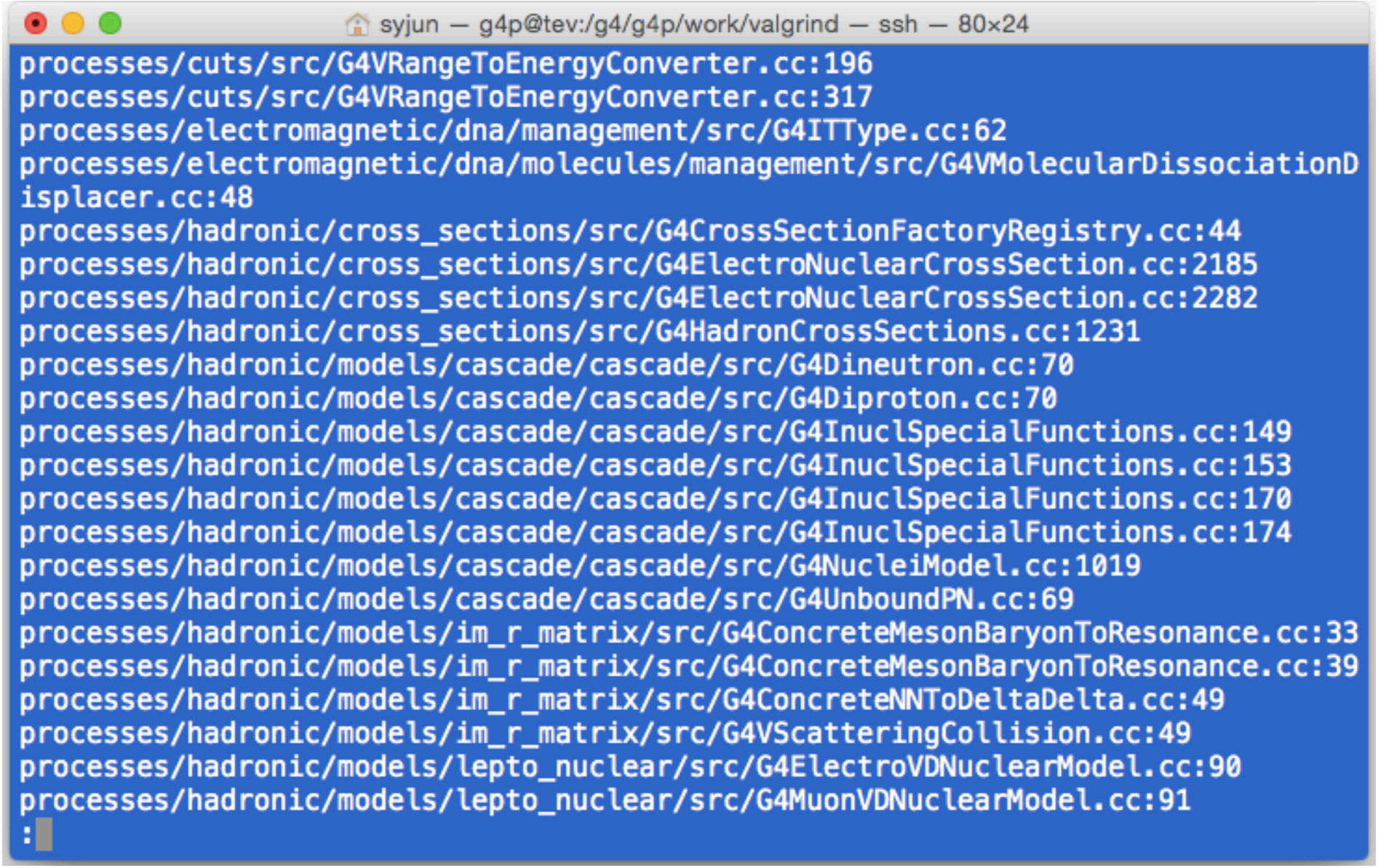
```
void* operator new(size_t size, const std::nothrow_t&) _NOEXCEPT
{
    return new(size, (char*)__builtin_return_address(0),0);
}
void operator delete(void* ptr) _NOEXCEPT
{
    delete(ptr, __builtin_return_address(0));
}
```

- Implementation of new and delete with binutils
  - builtin_return_address(0) : return address of the current function
  - addr2line(pointer) : convert the address of pointer to the file name and line number (only works for static libraries)
- For shared libraries, use a similar functionality of dladdr or dladdr1 in dlfunc.h (Dl_info)

Summary Report: A custom tool vs. Valgrind (exampleB2b)



List of Leaks: 125 from exampleB2b of 10.2.beta



Work Plan

- Run the memory leak monitor for each reference release
  - Select representative examples/tests
  - Post the list of potential leak (file names and line numbers)
  - Report a summary (and changes by the release version)
- Cross analysis with Valgrind outputs (each official release)
  - Correlation between entries (i.e, Valgrind vs. custom tool)

Summary

- Reviewed typical memory leak cases of Geant4
  - Object ownership (use C++11 smart pointers and move)
  - Elements of a container (std::vector<objects*>)
  - Shared objects for multi-threads (need a clear rule)
- Outputs from existing tools are complimentary to monitor potential memory leaks and mismanagements
  - IgProf
  - Valgrind
  - Coverity
- A new light-weight memory leak monitor will be deployed as a part of the benchmarking/profiling task
  - For each reference release
  - Provide the list of leaks found (file names and line numbers) and a summary