Findings and Recommendations pertaining to Elements of Geant4 Electromagnetic Code

J. Apostolakis[*], A. Dotti[†], K. L. Genser[‡,1], S. Y. Jun[‡], B. Norris[§]

[*]*CERN*
[†]*SLAC National Accelerator Laboratory*
[‡]*Fermi National Accelerator Laboratory*
[§]*Argonne National Laboratory, now at University of Oregon*

August 18th, 2014
Revised: October 22nd, 2014

This work was supported in part by US Department of Energy including through a joint High Energy Physics (HEP) and Advanced Scientific Computing Research (ASCR) Collaboration.

Abstract
Results of a review of computational aspects of Geant4 electromagnetic package and related code are presented.

Contents

[1]Review team coordinator

1 Introduction

1.1 Intended audience

This report is intended to be read by the authors and maintainers of the electromagnetic code of Geant4 as well as by other members of the Geant4 collaboration who may benefit from some of the more general comments pertinent to other Geant4 areas.

1.2 Scope of review

The electromagnetic (EM) processes package has a relatively high computational significance (CPU cost as a fraction of the total CPU time of a job) in most Geant4 application domains, including the detector simulation of HEP experiments. Some of the functions of the EM package are top contributors in CPU time profiles, placing in the top 10 most time-consuming by self time (considering only the exclusive time spent in the function itself, not its callees.) For these reasons it was decided to inspect the package with the main goal to assess if there are improvements which can be found to make it computationally more efficient.

The review concentrated on the most costly EM classes and functions. In some cases, other classes or functions which were heavily used by the EM code were also reviewed.

It was also considered that Geant4 code needs to be performant both in sequential and Multi-Threading modes and as easy as possible to read and maintain.

1.3 Review time-line and reviewed versions of Geant4

The review started in June of 2013 looking at a release near to the Geant4 10.0-beta public development release - in particular at Geant4 v9.6.r07 reference release which included some fixes, some consolidation and a small number of other improvements on top of 10.0-beta . It was effectively completed in June 2014 after Geant4 v10.1-beta was released.

Most comments of the review pertain to Geant4 v10. Intermediate review results were presented at the Geant4 Collaboration Meeting in September 2013 and at a joint meeting of the EM groups in March 2014. The code authors were being informed of the important findings with a significant computational impact as they were discovered so that they could be incorporated in the code development even before this report was completed.

1.4 Outline

In the next chapter (2), we identify the classes under review, in the EM package and beyond, and explain their place in the hierarchy of classes in the package. In chapter 3 we identify the environment and tools used to measure the contribution of functions, and the characteristics reported by these tools. In chapter 4 we discuss general issues, common between different functions, in particular the use of pseudo-random number generators. Chapter 5 contains the specific observations about different functions, in particular the most time consuming functions of key classes. Chapter 6 provides a summary of the measurements using the TAU [1] profiling tool, and our analysis of these results. Chapter 7 contains our observations on inlining, compiler optimization, and observations on the stability of code modifications. General recommendations are in chapter 8 and a summary in chapter 9. We also provide some suggestions for further review in the final chapter, number 10.

We note that the chapters 5 and 6 are quite detailed in analysis and recommendations and are intended for the code maintainers, whereas readers interested in more general recommendations and summary should read chapters 8 and 9.

2 Classes related to Geant4 EM Code

2.1 List of classes and functions under review

Table 1 shows the initial list of classes and functions which were to be reviewed as well as those which were added later. The list was obtained by profiling the `SimplifiedCalo` application as used in the official Geant4 Benchmarking and Profiling [2] (also see Chapter 3). This application was chosen as its geometry is simple, and the contributions of EM functions are larger, more stable and thus easier to measure.

2.2 The `G4VProcess` and `G4VEMModel` class hierarchies

Many of the classes in the table 1 belong to the Geant4 `G4VProcess` and `G4VEMModel` class hierarchies as shown in figures 1 and 2.

2.3 Other classes and functions

The `G4PhysicsVector` and `G4Physics2DVector` container classes and their descendants, although of a more general nature, are heavily used in the EM code and therefore were reviewed as well. Similarly, `G4Poisson`, a global function often used by the EM code was also reviewed.

3 Testing environment and tools

Performance measurements are essential in evaluating computing performance of applications, identifying problems and opportunities for code improvement and optimization. The set of software tools used in the performance evaluation procedure during the review include FAST [3], IgProf [4], TAU and Gooda [5]. FAST and IgProf are light weighted

Class name		Fractional CPU cost in %			
Function name	Geant4 version:	9.6.p02	9.6.r07	10.0.p01	10.0.r02
	gcc version:	4.4.6	4.4.6	4.4.6	4.8.2
G4PhysicsVector					
ComputeValue (in v10 renamed to Value)		4.74	5.04	9.18	9.18
G4PhysicsLogVector					
FindBinLocation		0.84	3.48	inlined	inlined
G4VProcess					
SubtractNumberOfInteractionLengthLeft		0.61	0.73	inlined	inlined
G4VEmProcess					
PostStepGetPhysicalInteractionLength		1.60	1.51	2.47	2.91
GetCurrentLambda		0.65	inlined	0.50	inlined
PostStepDoIt		0.79	0.62	0.67	0.72
G4VMultipleScattering					
AlongStepDoIt		0.69	0.81	0.73	0.73
AlongStepGetPhysicalInteractionLength		0.55	0.55	0.54	0.46
G4VEnergyLossProcess					
PostStepGetPhysicalInteractionLength		1.38	1.58	2.60	2.71
AlongStepDoIt		0.61	$\simeq$.50	0.59	0.65
GetLambdaForScaledEnergy		0.50	inlined	inlined	inlined
AlongStepGetPhysicalInteractionLength		$\simeq$.35	$\simeq$.40	0.43	0.47
G4VEmModel, G4VMscModel					
G4UrbanMscModel95 (v10 G4UrbanMscModel)					
ComputeGeomPathLength		0.80	1.02	0.90	1.17
ComputeTruePathLengthLimit		1.09	0.99	1.17	1.02
SampleCosineTheta		0.70	0.59	1.94	2.47
SampleScattering		$\simeq$.24	$\simeq$.31	0.58	0.47
Classes and functions added during the review (below)					
G4Physics2DVector					
Value		$\simeq$.30	$\simeq$.32	0.38	0.36
FindBinLocation		$\simeq$.20	$\simeq$.21	0.17	0.18
G4Poisson		inlined	0.81	inlined	inlined
G4UniversalFluctuation					
SampleFluctuations		2.76	1.85	4.01	4.28

Table 1: The list of classes and functions considered in the review, noting which were part of the initial set (above) and which were added during the review (below). Displayed is the fractional (exclusive) CPU cost of each function in percent of total, comparing also with the contemporaneous production releases, Geant4 v9.6.p02 and v10.p01. The data is based on SimplifiedCalo benchmark, configured with a beam of 50 GeV electrons, the FTFP_BERT physics list and no magnetic field. Values with $\simeq$ are extrapolations estimated based on the relative exclusive fractional cost of the associated function and may have a large systematic uncertainty, while other numbers are usually accurate within 0.1%. The first three versions of Geant4 were compiled with gcc 4.4.6, the last one with gcc 4.8.2

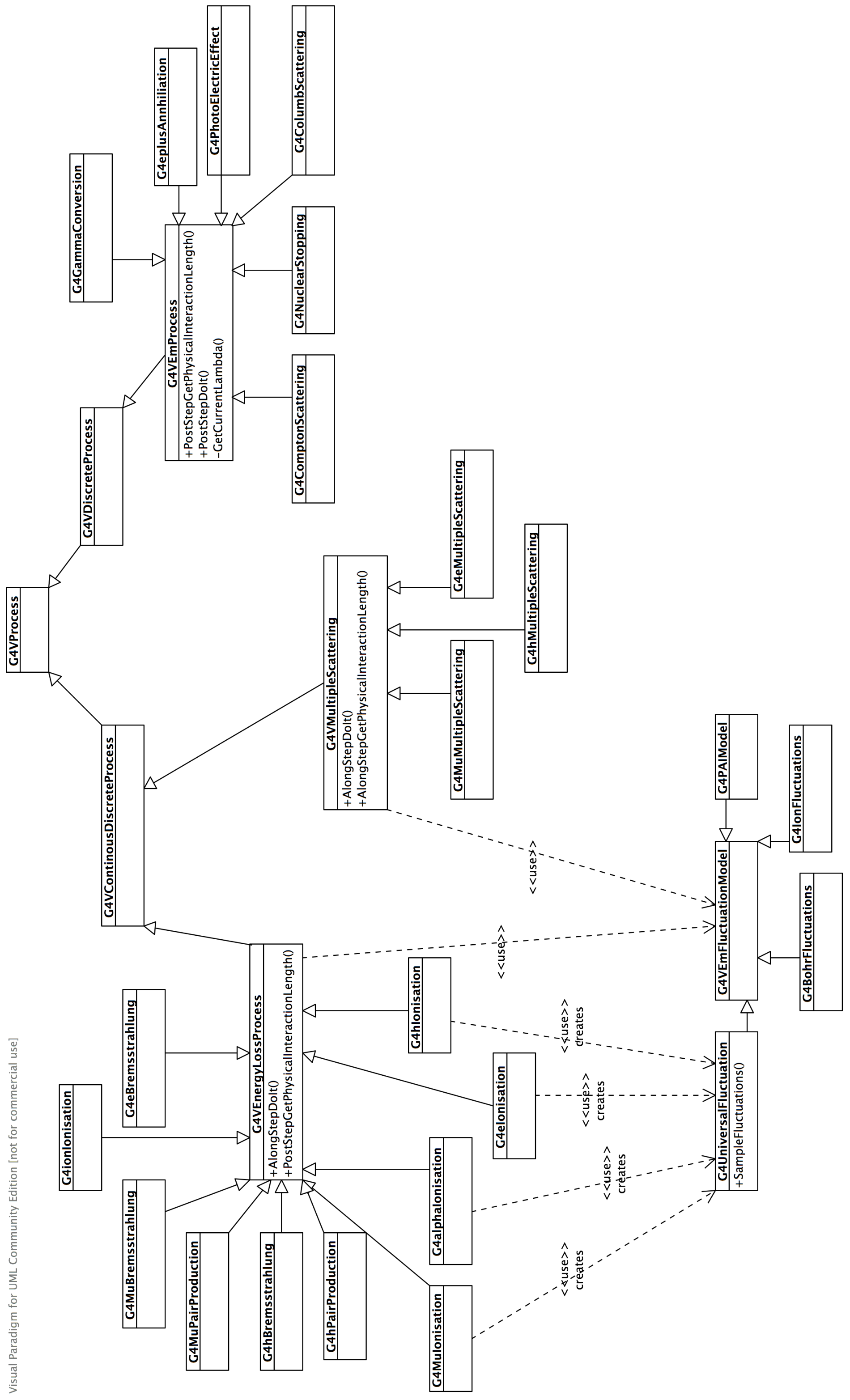

Figure 1: The G4VProcess class hierarchy and associated auxiliary classes

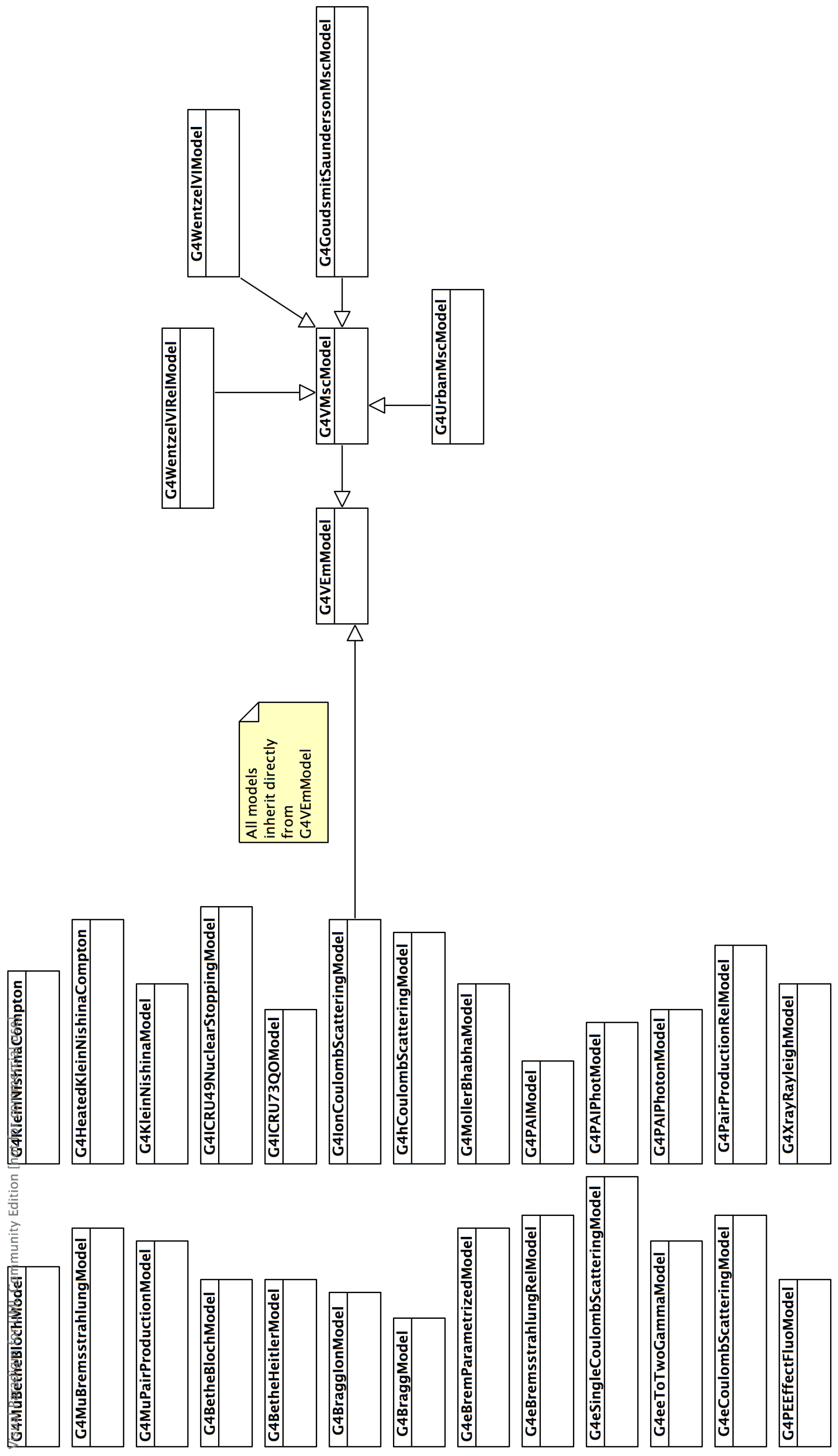

Figure 2: The G4VEMModel class hierarchy

sampling profilers while TAU is used for source based instrumentation with a relatively larger time overhead. Preliminary code changes were tested with the aforementioned Geant4 benchmark application (`SimplifiedCalo`), tracking 50 GeV electrons without magnetic field, to perform an initial evaluation. Subsequently, the recommended code was fully profiled with a set of applications that utilize different input event samples, physics lists, and detector configurations to collect statistically significant and comprehensive measurements.

FAST was used primarily for profiling overall CPU performance, while more detailed performance data with additional hardware counters were collected and analyzed using TAU. The time measurements are quoted for the case of using `SimplifiedCalo` with PYTHIA [6] Higgs events ($H \to ZZ, Z \to$ all decay channels) unless otherwise noted. The CPU and memory performance profiling using FAST and IgProf were done on nodes each equipped with 32-core AMD Opteron($^{\text{TM}}$) Processors 6128 (CPU: 2000 MHz, Cache: 512 KB). The gcc[7] version used in this review was 4.4.6 initially and 4.8.2 for the later tests. The most recent development version of TAU (2.23) available was used, incorporating improvements available until May 2014.

4 General comments

This chapter deals with items common between many different classes. The standout item in this regard is the use of random number generators. An additional item concerns the declaration of data members which have a constant value.

4.1 Use of random number generators

In many functions which we have reviewed, we observed that the calls to the pseudo-random number generators (pRNG) that are often done via macros, predominantly, if not exclusively, use functions returning single random numbers. In many cases, especially in loops, using the array interface would minimize the number of calls and would potentially allow more efficient generation of the random number sequences. For example, the following macro used in a loop:

```
#define G4UniformRand() G4MTHepRandom::getTheEngine()->flat()
```

can be replaced with a call to:

```
G4MTHepRandom::getTheEngine()->flatArray(const int size, double* vect)
```
[1]

followed by a loop iterating over the returned vector of random numbers. This revision is of most interest when the function is a significant consumer of CPU time, but the accumulated effect of many such changes in different code locations is also potentially relevant.

In functions that use the random number generator engine several times we suggest to cache its pointer value to remove the need to call the singleton pattern "getter" repeatedly. If this suggestion is followed, a way to propagate a change of pRNG engine to all clients is

[1]We tried inlining of `G4MTHepRandom::getTheEngine()` with a neutral effect.

likely required, to ensure that a change by the user of this engine between runs or in a start of run or start of event action is not missed.

We also noticed that the inverse of a random number is needed/used several times and therefore may warrant using/providing this type of interface. Other probability distribution functions are also used and required beyond the well established Gaussian and Poisson distribution. An optimized dedicated interface in the pRNG classes could be developed to take into consideration these additional cases.

Given the importance of random number generation, a review of its use in all Geant4 classes may be beneficial. See Section 5.10 for an example of suggested code transformations.

4.2 Constants and parameters with a constant value

A number of different classes contain several physical constants defined as member variables (data members). Other data members are parameters of the method which are never changed.

When these values are true constants or simply are never changed, it is good practice to 'promote' them into constants. In this way any future code maintenance which causes the value to be changed in the source code will be flagged by the compiler as an error - as it should be.

We note also that an advanced compiler can include the value of constants directly in expressions, and potentially do some computations at compile time - avoiding the need to do them at runtime.

One example of this issue in `G4VEnergyLossProcess` is data members such as e.g., `fMigdalConstant`, `fLPMconstant`, `klpm`, `kp`) could become either constant data members or local constants (or could declared as **`static const`** in the code near their use to improve clarity (by reducing the number of code locations the maintainer or user has to read, and the distance between them.)

4.3 Naming conventions for data members and global variables

The inspection of classes was hampered by the lack of some standard coding conventions.

In particular there is a strong need for a convention in the naming of data members. The lack of such a convention makes it harder to read (and thus also maintain) any non-trivial class or code. For example, even a reader who has inspected the code before and even made modifications, needs to carefully check whether particular variables are data members or not.

4.4 Improving constructors to avoid unwanted type conversions

A number of class constructor have either one argument constructors, or constructors with more arguments which have default arguments for all but the first argument. Such constructors could be used for implicit type conversion by a compiler. To protect against this we suggest to make it common practice that they are always declared explicit.

Example of such constructors are `G4VEmModel(const G4String& nam);` and `G4VEmProcess(const G4String&, G4ProcessType type = fElectromagnetic)` which can be be made explicit without causing any overhead or issue.

Although in some cases the omission of 'explicit' is unlikely to dangerous, we suggest to make its use routine for such constructors. This way it will be present in the cases in which it is an important safeguard, e.g. in utility classes which are reused within the EM package.

In addition we suggest to move the “initialization” assignment statements of constructors,

into the initializer list when possible (use parent class constructors and ternary operator if needed). An example which would benefit from this is the above G4VEmProcess constructor.

5 Class-specific observations

What follows are some class-specific observations. Is not an exhaustive list of comments one could make about the code we reviewed. Instead, we concentrated on making remarks with predominantly performance impact or ones related to the code correctness, with only occasional comments related to the programming style.

5.1 `G4VEnergyLossProcess`

The class `G4VEnergyLossProcess` is derived from `G4VContinuousDiscreteProcess` and all energy loss processes inherit from it.

While investigating classes inheriting from `G4VEnergyLossProcess`, we found that none of the derived classes except `G4eIonization` and `G4IonIonization` invoke `AlongStepGetPhysicalInteractionLength` and `AlongStepDoIt` during simulation.

The way this is done is the following: The `isIonisation` flag is set to `true` in the constructor of `G4VEnergyLossProcess`, but it is overridden by `SetIonisation(false)` in the constructors of the classes (or their parent classes) listed below:

- `G4eBremsstrahlung`
- `G4MuBremsstrahlung`

- `G4MuPairProduction`
- `G4ePolarizedBremsstrahlung`
- `G4hBremsstrahlung`
- `G4hPairProduction`

The class hierarchy can probably be optimized by reorganizing, and possibly absorbing the functionality of the following member functions of `G4VEnergyLossProcess` in the parent classes:

- `PostStepGetPhysicalInteractionLength`
  This member function overrides the original member function of `G4VContinuousDiscreteProcess` since a set of correction factors, including the scaled energy, are additionally applied in the case of the energy loss process. Nonetheless, these scale factors can be taken into account in other relevant places and the additional virtual function may not be necessary. For example, the scaled energy could be calculated when physics tables are built during the initialization and then it would not need to be evaluated in every stepping action for the default process (using the look-up table).
- `PostStepDoIt`
  Because this pure abstract function is implemented in the third layer of the class hierarchy for the first time, it may be worth to either remove the (intermediate) class, `G4VContinuousDiscreteProcess` or implement the common functionality in it.

The commonality of these two items leads us to suggest removing one class layer - we believe it would be possible to avoid inheriting from `G4VContinuousDiscreteProcess`, which anyway leads to confusion (see the next two sections.)

Regarding `PostStepGetPhysicalInteractionLength` we suggest to move the invocation of the bias manager out of this function to avoid the testing if it needs to be called in every stepping action for the default process.

There are places where the number of lines of code could be reduced. For a example, in member functions of the `G4VModel` classes called from `G4VEnergyLossProcess`, the following lines are redundant:

```
void G4eBremsstrahlungRelModel::SampleSecondaries( ... )
{
  ...
  kinEnergy   = kineticEnergy;
  totalEnergy = kineticEnergy + particleMass;
  densityCorr = densityFactor*totalEnergy*totalEnergy;
  ...
}
```

since they are already calculated in `G4eBremsstrahlungRelModel::SetupForMaterial`.

It seems that many data members in this class are introduced for the purpose of caching or sharing states across member functions which may be reduced by passing them as function arguments or introducing a simple helper class.

There appear to exist a few variables, such as `klpm` and `kp`, that are not used anywhere, but are declared and evaluated (i.e. a value is computed for them.)

It is also recommended that the data members should be reordered in the way that the most often used ones come first and big objects come later in the class so that the data is cached more efficiently (cf. ref. [8]).

5.2 `G4VContinuousDiscreteProcess` vs. `G4VDiscreteProcess`

Many of the EM classes derive from `G4VContinuousDiscreteProcess` and `G4VDiscreteProcess` (which are derived from `G4VProcess`).

Most electron processes inherit from `G4VContinuousDiscreteProcess` and yet behave as `G4VDiscreteProcess`: they override the `AlongStepDoIt` function, and if a flag is set they do nothing. In addition, in order to ensure that those “do nothing” functions are not executed, flags are utilized in the registration of these processes in the `G4ProcessManager` to ensure that these processes are treated as `G4VDiscreteProcess` i.e., that they do not appear in the list of processes which are called during the `AlongStep` loops, either for the interaction length polling or the process application (`DoIt`).

This was handled originally in the process of registering these classes in a physics list or a physics constructor. Now this task is undertaken by the `G4PhysicsListHelper` class which supersedes the process-specific class inheritance.

It appears that this solution has been chosen in order to have all the relevant classes inherit from one `G4VEnergyLossProcess` class, in order to share the energy loss and interaction length computation and associated tables and derived quantities. The current solution involves a rather obscure procedure of not registering a process for its `AlongStep` component. This is quite likely to be forgotten by future maintainers of the code. For this reason we believe it is important to consider whether an alternative solution (design) can achieve the same result, while being simpler to maintain.

If an alternative is not found, we strongly suggest that the choice of this solution is documented, and the requirements to keep it working made explicit. Since failing to maintain this “non-registration” would result in extra work and likely performance degradation, the code should either report a fatal exception or write a significant error which can be found during testing.

As explained earlier and in the section 5.1, most of electron processes inherited from `G4VContinuousDiscreteProcess` should probably be `G4VDiscreteProcess`.

It is also not clear whether `G4VDiscreteProcess` is an essential layer of the class inheritance hierarchy for photon processes and all of its member functions could probably be moved to `G4VEmProcess`. Reviewing what needs to be reimplemented, restructuring or possibly removing `G4VContinuousDiscreteProcess` and `G4VDiscreteProcess` may be explored in the future design.

5.3 `G4VEmProcess` vs. `G4VEnergyLossProcess`

We noted some code similarities between `G4VEnergyLossProcess` and `G4VEmProcess` e.g., in `AlongStepDoIt` or `PostStepDoIt`.

It looks like the `SelectModel` used in `PostStepDoIt` in both of the classes have the same functionality implemented in two different ways. It is not the only example of very similar functions. `SelectModelForMaterial` is another one. `DefineMaterial` are very similar.

It may seem that `G4VEmProcess` and `G4VEnergyLossProcess` were "derived" from a common code source and evolved together in a similar but not exactly the same way in many cases having the same functionality which could be factored out into another class or global functions. This observation serves as another argument that it may be possible to optimize the class structure/hierarchy.

5.4 `G4VEmProcess`

`PostStepGetPhysicalInteractionLength` is a virtual function, declared pure virtual at `G4VProcess` level, defined at the `G4VDiscreteProcess` level and redefined in `G4VEmProcess`. It does not perform many calculations, but is called many times what may be the reason why it takes a lot of CPU cycles. Based on the TAU analysis the derived class responsible for most invocations of the `PostStepGetPhysicalInteractionLength` is `G4CoulombScattering` (29.1%), followed closely by `G4GammaConversion`, `G4PhotoElectricEffect`, and `G4ComptonScattering` with 22.44% each (complete details are in Section 6.1).

We observed that `GetCurrentLambda` is a very short function which is declared inline; gcc 4.4.6 does not inline it, whereas 4.8.2 does.

We note relatively large Level2 data cache/instruction misses attributed to `PostStepGetPhysicalInteractionLength` and `PostStepDoIt` which may be another reason why the functions takes a lot of CPU time.

5.5 `G4VMultipleScattering`

In the current design in the `G4VProcess` hierarchy there is a class `G4VMultipleScattering` from which three concrete classes derive: `G4eMultipleScattering`, `G4hMultipleScattering` and `G4MuMultipleScattering`. The difference between the first and two later concrete classes (electron vs. hadron and muon) is very small, consisting of the choice of one parameter (`SetStepLimitType(fMinimal)` in `G4(h/Mu)MultipleScattering` functions, (plus a minimal printout difference). Whether this warrants a separate class is debatable, therefore we suggest merging the code if feasible.

5.6 G4PhysicsVector class hierarchy

We noted that the three main data members of this class: dataVector, binVector and secDerivative of std::vector<G4double> type, are usually used in "lockstep" in a loop. We then investigated if collocating the data which is used together would lead to a CPU time gain.

We replaced the above three data members with one, of std::vector<G4xyd> type, where G4xyd is a helper class with three G4double data members, **operator**<, and default (G4xyd()) and three argument G4xyd(G4double, G4double, G4double) constructors, i.e., we replaced three vectors of G4doubles with one vector of structs to localize access. We also initialized all data members in the constructors, using initializer list when possible, removed copy constructor and (assignment) **operator**= as the default ones supplied by the compiler should be correct eliminating the need to maintain the hand written code. We then modified derived classes accordingly. We also replaced some **if** statements with the ?: (ternary) operator and replaced hand-coded binary search in G4LPhysicsFreeVector::FindBinLocation with std::lower_bound

The overall effect of transforming G4PhysicsVector in Geant4 v9.6.r07 was about 1.5% performance improvement. For a comparison we note that the G4PhysicsVector::Value function itself used to take about 3% of the total execution time. We observed that the timing improvement was not attributed to this function itself. It occurred in other areas, mainly in CLHEP::MTwistEngine::flat, likely due to caching effects.

No measurable CPU effect was observed in Geant4 v10 for the equivalent transformation of the three main data members. Probably other code transformations which occurred between the versions impacted the above result. The code analysis using TAU revealed that G4PhysicsVector::Value() did become faster and had less Level2 cache misses, but other functions became slower.

We recommend to make the data members (containers) of G4PhysicsVector "read only" (and initialize them in the constructors), in order to conform with the best-practice 'rule' that shared objects in multi-threading must be constant. A design change may be required to cope with the potential lifecycles of the vector, e.g. whether its values are calculated by a particular class, or are retrieved from a file. This would also require a rewrite of the class G4PhysicsOrderedFreeVector which currently derives from G4PhysicsVector. Potentially this could become an independent class as it modifies its data members after the instantiation.

FindBinLocation member function

We noted that in Geant4 v10, FindBinLocation was moved to the base class, made non-virtual and inlined. An **if** was used to detect the type (using an **enum** G4PhysicsVectorType) of the underlying classes). After inlining FindBinLocation the G4PhysicsVector::Value function by itself took about 6% of the total execution time. In order to asses the effect of the inlining of FindBinLocation we profiled Geant4 v10 and found that the inlining causes about 1% performance degradation although there are event samples where it does help. Therefore the conclusion regarding the impact of the change is not as strong as in other cases.

Given that inlining of `FindBinLocation` may not be as beneficial as intended, we recommend abandoning the use of `G4PhysicsVectorType` and going back to using the virtual functions mechanism, delegating the type detection back to the compiler.

We also noted a potential typo in one of the type **`enum`**s in `G4PhysicsLnVector` where `T_G4PhysicsLogVector` is mixed with `T_G4PhysicsLnVector`. Using virtual functions avoids this type of problems.

5.7 G4Physics2DVector

`G4Physics2DVector` is another frequently used container class. We investigated changing the type of its underlying container to **`float`**. In unit tests, it resulted in about 13% degradation (probably due to the type conversion). [2]

We also investigated removing a level of indirection and attempted to provide for a decrease of the code maintenance by changing the underlying container type from
`std::vector<std::vector<G4double>*>` to
`std::vector<std::vector<G4double> >`
which allowed to remove the copy constructor, the assignment operator and the destructor. We also used `std::lower_bound` in `FindBinLocation` and inlined the function. Based on about 13% improvement in a unit test, we expected a minimal overall performance improvement, but the result was more than 5% degradation despite `G4Physics2DVector::Value` function taking only about 0.3% of the execution time, likely due to cache effects. In order to explain the slowdown effect we inspected the gcc Standard Template Library vector implementation and did not find any partial specialization for pointers to objects.

The above result suggests that this is one of those cases in which the data layout is more important than the algorithms. It underscores the importance of profiling/benchmarking after each code change as some of the profiling results may seem counter intuitive.

5.8 G4Poisson

`G4Poisson` is a frequently used global function. We noted that its code could be improved slightly, e.g., one should use correct types of the numeric literals, use preincrement instead of the postincrement operator, use the ternary operator instead of the last two lines, possibly cache the pointer to the random number engine, but more importantly, by just making the function inline one gains more than 30% CPU time in a unit test and about 2% overall.

[2] It resulted in about 12% improvement on an NVIDIA(^{TM}) GPU though.

After inlining of `G4Poisson`, the functions where the code was inlined became slightly more compute intensive, as it was expected, but it was more than offset by the gains in other functions, e.g., steppers.

5.9 G4UrbanMscModel

The inspection of the `ComputeTruePathLengthLimit` member function was made harder by its length and the three modes which it handles. We suggest separating some of the per-mode code into separate functions, and inline if possible. There was one passage of code that appeared to be completely common between the three different modes, apparently handling the cases of the first step in a volume or a boundary:

```
if(firstStep || stepStatus == fGeomBoundary) {
   G4double temptlimit = tlimit;
   if(temptlimit > tlimitmin)
   {
     do {
       temptlimit = G4RandGauss::shoot(tlimit,0.3*tlimit);
     } while ((temptlimit < tlimitmin) ||
              (temptlimit > 2.*tlimit-tlimitmin));
   }
   else { temptlimit = tlimitmin; }
   tPathLength = min(tPathLength, temptlimit);
```

We suggest to move this common code out of the `if/else if/else` in order to reduce the length of the code and the size of the resulting instructions.

In the member function `SampleScattering` several random numbers are used. In a manner similar to `G4UniversalFluctuation::SampleFluctuations` (see Sec. 5.10) a benefit can be obtained by caching the pointer to the pRNG and requesting a certain number of values needed in this function in one call using the pRNG array interface.

Also, in one of two cases a set of expensive trigonometric functions are used in order to calculate the sine and cosine of a compound angle:

```
if(std::abs(r*sth) < latcorr)
  Phi  = twopi*G4UniformRand();
else
{
     G4double psi = std::acos(latcorr/(r*sth));
     if(G4UniformRand() < 0.5)
        Phi = phi+psi;
     else
       Phi = phi-psi;
}

dirx = std::cos(Phi);
diry = std::sin(Phi);
```

In case this **else** branch is frequently used, this code can be revised to use only one expensive operation (square root) in place of the inverse-cosine, the sine and the cosine, by using the standard trigonometric equalities for the sine and cosine of the sum of two angles. In particular this can be done by computing the sine of `psi` from the cosine (including a sign in a similar way using a uniform random number), and then calculating the sine and cosine of `Phi` using the known relations for the sine and cosine of the addition of `Phi` and `psi`.

If the values of the sine and cosine of `Phi` (already calculated above) are stored in variables `sinPhi` and `cosPhi`, the resulting code would look something like the following:

```
if(std::abs(r*sth) < latcorr)
{
    G4double newPhi  = twopi*G4UniformRand();
    dirx = std::cos(newPhi);
    diry = std::sin(newPhi);
}
else
{
    G4double cosPsi= latcorr/(r*sth);
    G4double sinPsi= std::sqrt( 1.0 - cosPsi * cosPsi );
    if(G4UniformRand() < 0.5)
         sinPsi = - sinPsi;

    dirx= cosPhi * cosPsi - sinPhi * sinPsi;
    diry= sinPhi * cosPsi + cosPhi * sinPsi;
}
```

In addition to the above, we investigated inlining several of the shorter functions of this class, four of which were inlined by gcc v4.4.6 in Geant4 v10:

- `SampleDisplacement()`,
- `ComputeTheta0(G4double, G4double)`,
- `SimpleScattering(G4double, G4double)`,
- `LatCorrelation()`

leading to about 1.5% CPU improvement (comparable to the original fractional time spent in those functions). However, when the same code was inspected in Geant4 v10.ref01, only the first three functions were inlined by the compiler and there was no noticeable CPU time difference between the inlined and the non-inlined code.

5.10 `G4UniversalFluctuation::SampleFluctuations`

This function is one of the most CPU consuming. It does not call other functions except the random number generator functions. This makes it a strong target for improvement.

Profiling using Gooda [5] showed that a significant fraction of the execution time is spent in division of `double`s. An inspection of the code revealed that a number of divisions could be avoided, either by preparing the reciprocal or by doing a comparison in a different way. Proposed changes which took out three divisions were fed back to the code developers and adopted by them.

Another opportunity for optimization is the use of random numbers. The following line is responsible of a significant fraction of the relative CPU time:

```
for (G4int k=0; k<nb; ++k) { lossc += w2/(1.-w*G4UniformRand()); }
```

The loop has an upper limit `nb`, a random number itself, distributed according to the Poisson distribution. The calls to the random number generation inside the loop, create an unnecessary overhead; it also prevents the possible use of the array interface available for random number generators. Moreover, the call to retrieve the pointer to the random number generator in a multi-threaded build of Geant4 is even more expensive because it requires dealing with a Thread Local Storage pointer (also see comments in Sec. 4.1.)

We suggest to write the code as follows (see the code listing on page 19): First, retrieve the pointer to the (thread–local) instance of the random number engine only once and cache it to generate random numbers. Use the array interface. For the loop under discussion, use a static instance of the array to avoid re-creating the buffer to store the array of random numbers. Grow the array if needed.

With these modifications we have measured a 10% relative improvement of the `SampleFluctuations` function in a unit test. To make the code more readable we have tried to replace the code dealing with `rns` with the creation of a local variable in the function. Such a simplification, however, reduces the CPU gain (we have also tried to use of `std::vector` instead of a c-array, but in such a case the code was found to have slowed down).

Finally we have noticed that the algorithm of sampling fluctuations is contained in the following `for` loop: `for (G4int istep=0; istep < nstep; ++istep) {...}` where the `nstep` upper limit can take the values of 1 or 2. To take advantage of such property we have tried to remove the loop and apply several code transformations. Unfortunately, these have not introduced any further CPU benefit and instead have made the code less readable. We do not thus suggest further transformations.

6 Observations based on TAU measurements and analysis

The measurement and associated analyzes were conducted throughout the review and the results were used to understand the effects of attempted code transformations (e.g., the data structure modification in Sec. 5.6). In addition, the measurement infrastructure developed during the review included a number of performance tool extensions and initial automation that will make detailed performance analysis in the future less effort-intensive and more efficient.

We performed detailed measurements with the TAU Performance System® [1], creating a database of results by using the performance data management system, *TAUdb* [16, 15]. TAUdb stores profiles from performance experiments along with metadata describing the execution environment and application-specific metadata. Data stored in TAUdb is

```
//Create thread-local cache object to store pRNG numbers.
//Use a reasonable size for the array to keep pRNG.
namespace {
        G4ThreadLocal int siz = 30;
        G4ThreadLocal double* rns = 0;
        G4ThreadLocal CLHEP::HepRandomEngine* rngEngine = 0;
}

G4double G4UniversalFluctuation::SampleFluctuations(...) {
        ...
        // First time that the pRNG is accessed
        if ( !rngEngine ) rngEngine = G4MTHepRandom::getTheEngine();

        // Note that in the following we use always rngEngine. For
        // example we use G4RandGauss::shoot( rngEngine, mean, sigma)
        // instead of G4RandGaus::shoot(mean,sigma)
        ...
        const G4int nb = G4Poisson(p3);
        if ( nb > siz || !rns )
        {
            delete[] rns;
            siz=nb;
            rns = new double[nb]; // use unique_ptr in future
        }
        if(nb > 0) {
            rngEngine->flatArray(nb,rns);
            for (G4int k=0; k<nb; ++k) { lossc += w2/(1.-w*rns[k]);}
        }
        ...
}
```

accessible from the *ParaProf* [11] parallel profile analyzer and the *PerfExplorer* [13, 14] performance data mining framework. PerfExplorer provides a library of internal analysis routines and an interface to statistics and data mining packages, such as R and Weka [17]. An API to the analysis library and a Python scripting engine are available so that analysis pipelines can be specified programatically.

We instrumented Geant4 with the TAU suite of tools using two different strategies. First, we performed a full instrumentation at the function level, which enabled us to collect accurate measurements of a variety of low-level hardware counters. Because the runtime overhead of this type of profiling is significant, we also implemented an alternate instrumentation approach that only added performance data collection to selected classes and functions listed in Table 1. The instrumented source code of Geant4 was then built by using the same build configuration as normal production runs. The instrumentation does not significantly perturb metrics such as cache misses, which is verified through comparison of instrumentation-based measurements with non-intrusive sampling-based results. We also verified through inspection of the object code that the instrumentation does not interfere with inlining, i.e., that functions specified as inlined are still being inlined after instrumentation.

The instrumentation and analysis of a large, complex C++ application such as Geant4 required a number of extensions in the TAU performance tools and resulted in the creation of new analysis capabilities, which will be usable in other application contexts. Examples of new capabilities include the following:

- Profiling of inlined functions;
- Identification of ”lightweight” classes, i.e., intermediate layer (semi-abstract) classes in the hierarchy that consume a significant percentage of resources while doing little useful work;
- Accumulation of different metrics (time, any performance hardware counter, or derived metrics) by *class*, sorting by the fraction of total.

To our knowledge, no other performance tool provides class-based statistics, which are important when one wishes to consider the performance impact of different object oriented design decisions.

6.1 `SimplifiedCalo` Results

We performed a number of experiments with the `SimplifiedCalo` benchmark code and Geant4 v. 10 to collect the hardware statistics described in Table 2. The tests were performed on the Aciss cluster at University of Oregon, specifically on Intel Xeon X5650 2.67GHz 12-core dual processor nodes with 70GB of DRAM (per node). Geant4 was compiled with gcc 4.8.2 using the RelWithDebInfo (with the default optimization level -O2) cmake build type.

To collect the performance counter data we used automated instrumentation of the source code through TAU compiler wrappers by configuring Geant4 with `-DCMAKE_CXX_-COMPILER=tau_cxx.sh -DCMAKE_C_COMPILER=tau_cc.sh`.

All of the data can be accessed by connecting to the TauDB database hosted at the University of Oregon (contact Boyana Norris for configuration details). We implemented several PerfExplorer analysis scripts to produce statistics grouped by classes of interest, sorting classes and their functions by the fraction of the total amount for each metric. The scripts are written in Jython and will be documented and made available online.

All runs were done with Geant4 v10 and used the standard electromagnetic physics list. The `SimplifiedCalo` settings used for the experiments are listed in Appendix A unless otherwise noted.

Measurements Summary

The following experimental data shows several performance counters (collected through instrumentation, unless indicated otherwise) for the classes in Table 1. We focus on

Category	Metric	Description
Time	P_WALL_CLOCK_TIME	Wall-clock time
Memory	L1_DCM	Level1 data cache misses
Memory	L2_DCM	Level2 data cache misses
Memory	L2_DCA	Level2 total number of Level2 data accesses
Memory	L1_ICM	Level1 instruction cache misses
Memory	L2_ICM	Level2 instruction cache misses
Memory	L2_TCM	Total Level2 cache misses (data and instructions)
Memory	L3_TCA	Total Level3 cache accesses
Memory	LD_INS	Load instructions
Memory	TLB_DM	Data translation lookaside buffer misses
Memory	TLB_IM	Instruction translation lookaside buffer misses
Memory	TLB_TL	Total translation lookaside buffer misses
Jumps	BR_MSP	Number of mispredicted branches
Jumps	BR_INS	Total number of branch instructions
Compute	DP_OPS	Number of double precision floating-point operations
Compute	FP_OPS	Number of single precision floating-point operations
General	VEC_DP	Vector/SIMD instructions executed (double precision)
General	VEC_SP	Vector/SIMD instructions executed (single precision)
General	TOT_CYC	Total cycles
General	TOT_IIS	Total instructions issued
General	TOT_INS	Total instructions executed
General	RES_STL	Total resource stalls (any reason)

Table 2: Hardware metrics collected for `SimplifiedCalo`.

metrics that are typically the main contributor to performance degradation. Data for functions marked with * was obtained with sampling and includes initialization. All other data was obtained with instrumentation and does *not* include initialization. At this time, headers could not be instrumented automatically and hence we relied on sampling for data for functions defined in headers. For instrumentation-based measurements, we show the inclusive measurement percent of total, followed in parentheses by the exclusive measurement as percent of total. An *inclusive* measurement includes the value of the counter (or time) for that function and all the functions called within it, while an *exclusive* measurement includes only the value of the counter (or time) for the function, excluding measurements for functions called within in.

The `G4PhysicsVector::Value` function was responsible for 8.5% of CPU stall cycles but only 3% of the instructions executed for the application (excluding initialization). Looking into it in more detail, we observed that it is responsible for over 4% of TLB [3] data misses and 2% of all Level2 cache misses (including 3% Level2 data and 0.9% Level2 instruction cache misses), which are likely the main causes of the stalls. To a lesser extent, the same relatively high stall rates caused mainly by TLB instruction misses can be seen in `G4UrbanMscModel`'s `ComputeTruePathLengthLimit` (this can be caused by aggressive inlining or loop unrolling). For `G4VEmProcess`'s `PostStepGetPhysicalInteractionLength` and `PostStepDoIt` functions, the stalls are caused by a combination of Level2 and TLB data and instruction misses. On the other hand, while `G4UniversalFluctuation`'s `SampleFluctuations` function incurs a large

[3]Translation Lookaside Buffer - a cache used to improve virtual address translation speed.

Class name	Performance hardware counter measurements (%Tot)				
Function name	Stalls	L2 TCM	TLB DM	TLB IM	Time
	Inc. (Excl.)	Inc. (Excl.)	Inc. (Excl.)	Inc. (Excl.)	Inc. (Excl.)
`G4PhysicsVector`					
`Value`	8.46 (8.46)	1.67 (1.67)	4.13 (4.13)	1.46 (1.46)	2.68 (2.68)
`G4PhysicsLogVector`					
`FindBinLocation`	–	–	–	–	–
`G4VProcess`					
`SubtractNumberOfInteractionLengthLeft`	–	–	–	–	–
`G4VEmProcess`					
`PostStepGetPhysicalInteractionLength`	4.07 (1.49)	2.20 (1.62)	3.09 (1.95)	1.93 (1.45)	2.51 (1.82)
`GetCurrentLambda`*	<0.01	<0.01	<0.01	<0.01	<0.01
`PostStepDoIt`	4.56 (0.68)	2.93 (1.31)	1.83 (0.64)	2.57 (1.16)	1.68 (0.72)
`G4VMultipleScattering`					
`AlongStepDoIt`*	2.1 (0.2)	1.9 (0.4)	–	1.7 (0.3)	1.8 (0.2)
`AlongStepGetPhysicalInteractionLength`*	4.8 (0.2)	4.2 (0.4)	–	2.6 (0.2)	2.3 (0.3)
`G4VEnergyLossProcess`					
`PostStepGetPhysicalInteractionLength`*	1.8 (0.4)	1.9 (1.0)	–	0.6 (0.5)	1.2 (0.7)
`AlongStepDoIt`*	2.8 (0.3)	1.0 (0.4)	–	0.8 (0.3)	4.2 (0.3)
`GetLambdaForScaledEnergy`	–	–	–	–	–
`AlongStepGetPhysicalInteractionLength`*	0.5 (0.2)	0.5 (0.2)	–	<0.1	0.4 (0.3)
`G4UrbanMscModel`					
`ComputeGeomPathLength`	1.04 (0.61)	0.56 (0.48)	0.78 (0.55)	1.93 (1.79)	0.62 (0.47)
`ComputeTruePathLengthLimit`	3.67 (0.99)	2.90 (1.99)	2.52 (0.91)	6.02 (3.78)	2.37 (1.19)
`SampleCosineTheta`	1.84 (1.78)	0.61 (0.58)	0.36 (0.21)	0.49 (0.48)	0.31 (0.21)
`SampleScattering`	3.20 (1.22)	1.32 (0.67)	0.77 (0.25)	1.18 (0.65)	0.73 (0.31)
Classes and functions added during the review					
`G4Physics2DVector`					
`Value`	0.40 (0.39)	0.36 (0.32)	0.40 (0.26)	0.03 (0.03)	0.34 (0.24)
`FindBinLocation`	<0.01	0.05 (0.05)	0.13 (0.13)	<0.01	0.01 (0.01)
`G4Poisson`*	<0.01	0.22	–	<0.01	0.33
`G4UniversalFluctuation`					
`SampleFluctuations`	8.10 (8.10)	0.53 (0.53)	0.18 (0.18)	1.10 (1.10)	0.13 (0.13)

Table 3: Hardware counter-based measurements shown as percent of total (for Geant4 v10). The functions for which no data is shown were not called in the experiments we performed while collecting these data. Data for functions marked with * was obtained with sampling and includes initialization. All other data was obtained with instrumentation and does *not* include initialization.

number of stalls over a short time, making it one of the most FLOP-inefficient functions as defined in the next section, it is not immediately evident why the stall rate is so high. More detailed analysis is needed to establish with certainty the dominant cause for the high stall cycle counts in this and some of the other functions.

We also computed a floating-point *inefficiency* metric which identifies functions that have high proportion stalled cycles (for any reason) and also perform a significant number of floating-point computation (as a fraction of all instructions). We computed this as the weighted rate

$$FP_{ineff} = w \frac{Ins_{fp}}{Ins_{total}} \frac{Cycles_{stall}}{Cycles_{total}}$$

where w is the function cycles (exclusive) fraction of total application cycles and Ins_{fp} is the number of floating-point instructions, and Ins_{total} is the total number of instructions.

Table 4 shows the most FLOP-inefficient functions among the ones we considered during the review (those, for which the metric represents more than 1% of the application total).

Class name Function name	FP_{ineff} Value (Excl.)	Number of Calls	% of Total
G4UniversalFluctuation			60
SampleFluctuations	1.25e+00	1.56e+07	60
G4UrbanMscModel			16
SampleCosineTheta	1.98e-01	1.44e+07	9.5
SampleScattering	9.31e-02	1.44e+07	4.5
ComputeTruePathLengthLimit	2.68e-02	4.57e+07	1.3
G4PhysicsVector			11.3
Value	2.35e-01	3.71e+08	11.3
G4VEmProcess			2.2
PostStepGetPhysicalInteractionLength	2.99e-02	1.61e+08	1.4
G4VEnergyLossProcess			1.8
PostStepGetPhysicalInteractionLength	3.35e-02	9.38e+07	1.6

Table 4: FLOP-inefficient functions that have high proportion stalled cycles (for any reason) and also perform a significant number of floating-point computation (as a fraction of all instructions) and are thus candidates for optimization.

Lightweight Functions

We identified "lightweight" functions, i.e., those functions that have relatively few floating-point instructions per call but take a significant fraction ($> 0.5\%$) of the overall execution time. These are grouped by class and summarized in Table 5. These results are based on data obtained through instrumentation (which has accurate function call counts). The FLOP-inefficient `G4VEmProcess::PostStepGetPhysicalInteractionLength` is also among the lightweight functions because each call does relatively few operations (but there are many calls).

Class Function	Total Instr.	Number of Calls	FP Instr. per Call	% of Total Time
G4VEmProcess	1.56e+10			2.97
PostStepGetPhysicalInteractionLength	1.14e+10	1.61e+08	70.64	1.82
PostStepDoIt	3.89e+09	3.05e+07	127.44	0.72

Table 5: Lightweight functions that have relatively few floating-point instructions per call but take a significant fraction ($> 0.5\%$) of the overall execution time.

PostStepDoIt and PostStepGetPhysicalInteractionLength

To understand better the use of these two significant functions implemented in `G4VEmProcess` (discussed in Section 5.4), we instrumented the implementation to record the specific types of the process objects for which these virtual functions are invoked among the subclasses of `G4VEmProcess` and count the number of calls made for each type. The separate instrumentation was necessary because these virtual functions are dynamically dispatched to the `G4VEmProcess` implementation of each function, and thus regular performance measurements show calls to `PostStepDoIt` and `PostStepGetPhysicalInteractionLength` as calls on `G4VEmProcess` instances.

The `PostStepDoIt` virtual function is called in the
`void G4SteppingManager::InvokePSDIP(size_t np)` function and in most instances this results in a call to the implementation provided in the base class `G4VEmProcess`. Tables 6 and 7 show a breakdown of the number of calls made through objects of each concrete type (These counts are for 500 events, the total number of calls was 1.386e+08).

Class name	Count	% of total
`G4ComptonScattering`	20668199	14.91%
`G4PhotoElectricEffect`	8916255	6.43%
`G4GammaConversion`	852255	0.62%
`G4eplusAnnihilation`	79927	0.06%
`G4CoulombScattering`	999	0.0007%

Table 6: Types of objects for which `PostStepDoIt` was invoked (in the `G4VEmProcess` hierarchy).

Class name	Count	% of total
`G4Transportation`	83117473	59.97%
`G4eMultipleScattering`	16078703	11.60%
`G4eBremsstrahlung`	8476856	6.12%
`G4eIonisation`	450700	0.32%

Table 7: Types of objects for which `PostStepDoIt` was invoked (not in the `G4VEmProcess` hierarchy).

The `PostStepGetPhysicalInteractionLength` virtual function is called in the `G4VProcess::PostStepGPIL` function and in most instances this results in a call to the implementation provided in the base class `G4VEmProcess`. Table 8 shows a breakdown of the number of calls made through objects of each concrete type (These counts are for 500 events, the total number of calls was 1.612476e+08).

Class name	Count	% of total
`G4CoulombScattering`	4.692927e+07	29.10%
`G4GammaConversion`	3.618821e+07	22.44%
`G4PhotoElectricEffect`	3.618821e+07	22.44%
`G4ComptonScattering`	3.618821e+07	22.44%
`G4eplusAnnihilation`	5.753727e+06	3.57%

Table 8: Types of objects for which `PostStepGetPhysicalInteractionLength` was invoked (in the `G4VEmProcess` hierarchy).

6.2 Results of `G4PhysicsVector` modifications using `G4xyd` helper class

Table 9 shows a comparison of the top functions (in version 10.0) sorted by *exclusive* execution times, with the timings and Level2 total cache misses for a version where `G4PhysicsVector` data members were replaced with the `std::vector<G4xyd>` as described in section 5.6 (denoted 10xyd in the table). The improvement percentage is computed as $(T_{v10} - T_{xyd})/T_{v10}$ and positive numbers indicate improvement, while negative values indicate degradation. Total Level2 misses (labeled with L2_TCM) were computed similarly and a positive value in the improvement column indicates improvement, while a negative value indicates degradation. This data was collected for the same experiment configuration as in the rest of this section. The measurements represent exclusive values (only for the function indicated) and were obtained with low-overhead sampling using a sampling period of 1,000 microseconds.

The replacement of the separate `dataVector`, `binVector`, and `secDerivative` vectors with a single vector of the new (combined) type `G4xyd` did not result in significant improvements of the `G4PhysicsVector::Value` computation (including the two functions it calls, `G4PhysicsVector::SplineInterpolation()` and `G4PhysicsVector::FindBinLocation()`). Note that `G4PhysicsVector::Value` calls `G4PhysicsVector::SplineInterpolation()` and `G4PhysicsVector::FindBinLocation()` – the combined exclusive times of these three functions represent the inclusive `Value` function time which is 7.1% and 7.3% of the total time for versions 10 and 10xyd, respectively. The overall effects were mixed, with degradation observed in some seemingly unrelated functions and an overall slowdown of 2.45%. The changes in the xyd version did improve the L2 cache misses by 5.3% overall, but that does not compensate for the performance degradation. Additional analysis is needed to determine the specific causes.

7 Impact of changes

7.1 Impact of inlining and using compiler optimization

We looked at the impact of inlining and optimization (when using gcc).

We observed that the optimized and inlined code is much faster (even by a factor of four for the full `SimplifiedCalo` at optimization levels of 2 and 3) compared to the non-optimized/non-inlined one. Due to the optimizations, even seemingly local code changes can modify the resulting final executable quite significantly (based on an inspection using GNU objdump).

The inlining should be considered with caution though, as at times it failed even for apparently suitable cases (short functions), and we have clearly demonstrated that it does not always improve the performance (cf. subsection 5.6). Inlining short functions and computationally critical code is usually a good approach, however one should always double check if the final result is beneficial from the performance point of view by profiling

Time (milliseconds); exclusive; sampling; includes initialization					Total L2 misses; Exclusive		
%total v10	%total xyd	Time (μs) v10	Time (μs) xyd	% Improv.	L2_TCM v10	L2_TCM xyd	% Improv.
Total		**226666**	**232211**	**-2.45%**	**2.38E+09**	**2.25E+09**	**5.37%**
`UNRESOLVED /lib64/libm-2.12.so`							
6.2%	5.4%	13940	12456	10.65%	1.47E+08	1.22E+08	16.83%
`4Log.hhG4Log [G]`							
4.4%	4.3%	10052	9886	1.65%	1.06E+08	9.71E+07	8.67%
`G4PhysicsVector::SplineInterpolation()`							
4.4%	4.3%	9967	9963	0.00%	1.05E+08	9.77E+07	7.09%
`G4SteppingManager::DefinePhysicalStepLength()`							
2.4%	2.3%	5422	5339	1.53%	5.70E+07	5.22E+07	8.46%
`CLHEP::Hep3Vector::`**`operator=`**`()`							
2.1%	2.2%	4755	4993	-0.50%	4.98E+07	4.86E+07	2.39%
`4Exp.hhG4Exp [G]`							
2.0%	2.0%	4597	4560	0.80%	4.89E+07	4.52E+07	7.55%
`G4UniversalFluctuation::SampleFluctuations()`							
1.9%	1.9%	4417	4395	0.49%	4.70E+07	4.37E+07	6.98%
`CLHEP::MTwistEngine::flat()`							
1.7%	2.0%	3876	4575	-18.03%	4.10E+07	4.50E+07	-9.91%
`G4PhysicsVector::FindBinLocation()`							
1.7%	1.8%	3766	4240	-12.59%	3.97E+07	4.15E+07	-4.56%
`G4Transportation::AlongStepGetPhysicalInteractionLength()`							
1.4%	0.9%	3255	2191	32.69%	3.42E+07	2.13E+07	37.65%
`G4Navigator::ComputeStep()`							
1.4%	1.2%	3230	2870	11.15%	3.38E+07	2.80E+07	17.17%
`G4ParticleChange::CheckIt()`							
1.3%	1.3%	2992	3072	-2.67%	3.14E+07	2.99E+07	4.63%
`CLHEP::Hep3Vector::rotateUz()`							
1.3%	1.3%	2945	2916	0.98%	3.11E+07	2.87E+07	7.94%
`G4NormalNavigation::ComputeStep()`							
1.2%	1.1%	2719	2576	5.26%	2.87E+07	2.52E+07	12.23%
`4Log.hhget_log_px [G]`							
1.2%	1.2%	2617	2693	-2.90%	2.77E+07	2.65E+07	4.30%
`G4VEmProcess::PostStepGetPhysicalInteractionLength()`							
1.1%	1.4%	2515	3225	-28.23%	2.64E+07	3.15E+07	-19.26%
`G4PhysicsVector::Value()`							
1.0%	1.2%	2353	2747	-16.74%	2.48E+07	2.67E+07	-7.78%
`G4SteppingManager::InvokeAlongStepDoItProcs()`							
1.0%	1.2%	2309	2672	-15.72%	2.42E+07	2.59E+07	-7.19%
`G4SteppingManager::Stepping()`							
1.0%	1.1%	2288	2495	-9.05%	2.40E+07	2.44E+07	-1.50%
`G4UrbanMscModel::ComputeGeomPathLength()`							
1.0%	0.8%	2287	1887	17.49%	2.40E+07	1.84E+07	23.16%
`G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength()`							
1.0%	1.0%	2251	2228	1.02%	2.37E+07	2.17E+07	8.32%
`G4AffineTransform::TransformPoint()`							
1.0%	0.9 %	2238	2154	3.75%	2.36E+07	2.09E+07	11.33%

Table 9: Sampling measurement comparison between versions 10.0 and 10xyd. See section 6.2 for related discussion.

and benchmarking the resulting executable. Using gcc `-Winline` option enables warnings when inlining fails (outside sysem headers).

7.2 Impact of code modifications

We point out that the increase of the fractional cost of most of the functions listed in Table 1 was due to inlining of other functions invoked by them.

We also note that the CPU impact of code changes may be bigger than the initial CPU cost of the code in question, likely due to caching effects (e.g., see section 5.7).

8 General recommendations

The following are recommendations of a more general nature pertaining to more than one area of the code.

To eliminate unnecessary code maintenance and to (likely) improve performance we suggest to eliminate custom written copy constructors, assignment operators and destructors when the compiler supplied ones are correct; use the ?: (ternary) operator when possible (beware of implicit conversion though); use Standard Library algorithms; e.g., `std::lower_bound` instead of hand written binary search.

Make all single argument constructors (and those which default to such) explicit, to avoid unintentional type conversions.

Adopt some consistent naming conventions e.g., for data members and member functions, to ease reading of the code.

Document and explain the motivation behind the non standard definition of the `operator==` and `operator!=` as they test for identity not equality (same address in memory vs. equal values), quite common pattern not only in the EM code, as the code may confuse future code maintainers.

Avoid, and if not possible, document well non obvious side effects and cross dependencies.

Based on the code we saw, we suggest that each code change should be ”approved” by another person (e.g., to be chosen by the author of the code change) to minimize chances to make mistakes and to increase the code quality. This should be in addition to the current proposed tag approval after the code compiles and passes the standard “code does not crash” test.

We do note that the above recommendations (and findings that lead to them) are not unique to the EM code only as e.g., indicated in the previous code reviews[10] and cursory inspection of other areas of the Geant4 code.

9 Summary

A review of compute intensive functions of the electromagnetic processes package (and some functions frequently used in there) of Geant4 was conducted. The code was inspected both visually and using TAU Performance System. It was also profiled with FAST and IgProf tools.

The review was performed in close collaboration with the code authors or maintainers who were being informed of the important findings with a significant computational impact as soon as they were discovered.

The review resulted in several recommendations with a computational impact and provided additional insight regarding cache misses and other hardware metrics. It also suggested a need for further code reviews in the areas indicated below.

The review process itself stimulated the development of the TAU package and shall have a positive impact on future reviews using TAU.

Most of the gains were obtained by code inlining. We note however that it needs to be done very judiciously as some of our inlining attempts did not result in a performance gain and had to be retracted.

This underscored the need to verify each code change by performing statistically significant profiling and benchmarking as some of the results may seem counter intuitive, likely due to the complicated interplay between the data structures/layout and the algorithms operating on them. Also, it is difficult to create a reliable prediction or model the factors which affect the performance of current hardware, due to its complex nature and the way in which the compiler harnesses the hardware.

Another review finding was the realization that some of the functions in the `G4VContinuousDiscreteProcess` and `G4VDiscreteProcess` class hierarchies are quite similar and some of the sub-classes behave in a not very intuitive way as a quite complicated procedure is used to activate certain functions, also suggesting that some of the sub-classes should probably be moved from one hierarchy to the other and/or the class structure could be optimized.

The total performance gain based on the review recommendations was about 3% although the exact number is difficult to obtain as it fluctuated depending on the Geant4 release and changes in other areas of the toolkit.

10 Suggestions for further review

During the work of this review we have identified other possible areas of the code that could benefit from a similar review procedure, therefore we suggest to review:

- Both random number generators and their use.
- The entire `G4VProcess` class hierarchy to see if it could be simplified.
- Adherance to coding conventions and good practices.

Regarding the last item, we recognize that automatic tools can largely simplify the task of reviewing large code bases in an efficient manner. Some of these tools are specifically designed to improve code quality: for example Coverity is already used by Geant4 collaboration to identify code defects. Coverity focus is to identify code lines that can be bugs or inefficiencies. We suggest the evaluation of another tool (ACRE) (`https://github.com/steinj/acre`, currently in prototype state) that instead is focused on verifying the implementation of good code practices that can substantially help code reviews. The tool is still under active development but it could be used to conduct code-reviews on large portion of Geant4 code in a semi-automated way.

11 Acknowledgments

We appreciate the advice and assistance received from Vladimir Ivanchenko, Laszlo Urban, Gabriele Cosmo, Daniel Elvira, Chris Green, Jim Kowalkowski and Marc Paterno.

Bibliography

[1] TAU (Tuning and Analysis Utilities)
http://www.cs.uoregon.edu/research/tau/home.php

[2] Geant4 Benchmarking and Profiling:
https://g4cpt.fnal.gov/perfanalysis/g4p/admin/task.html

[3] Flexible Analysis and Storage Toolkit (FAST)
https://cdcvs.fnal.gov/redmine/projects/fast/documents

[4] IgProf, The Ignominous Profiler
http://igprof.org/

[5] Gooda - emu event analysis package.
https://code.google.com/p/gooda/

[6] PYTHIA a program for the generation of high-energy physics events
http://home.thep.lu.se/~torbjorn/Pythia.html

[7] GCC, the GNU Compiler Collection
http://gcc.gnu.org

[8] Discussion benefits of re-ordering data members in classes
http://stackoverflow.com/questions/892767/c-optimizing-member-variable-order

[9] Coverity for CERN projects, http://sftweb.cern.ch/coverity

[10] Previous reviews of Geant4 CHIPS and Propagation in Fields packages available from: https://twiki.cern.ch/twiki/bin/view/Geant4/G4CPT

[11] R. Bell, A. Malony and S. Shende, *"A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis"*.

[12] University of Oregon Performance Research Lab, *"ParaProf – User's Manual"*.

[13] K. Huck and A. Malony, *"PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing"*.

[14] K. Huck, A. Malony, S. Shende and A. Morris, *"Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0"*, (special issue on Large-Scale Programming Tools and Environments).

[15] K. Huck, A. Malony, R. Bell and A. Morris, *"Design and implementation of a parallel performance data management framework"*, in: *"Parallel Processing, 2005. ICPP 2005. International Conference on"*, 473-482p.

[16] K. Huck, M. Millstein, A. Malony and S. Shende, *"TAUdb: PerfDMF Refactored"*, in: *"Performance Tools for Extreme-scale Computing"*, presented at CSCADS Summer Workshop.

[17] Hall, Mark and Frank, Eibe and Holmes, Geoffrey and Pfahringer, Bernhard and Reutemann, P eter and Witten, Ian H. *The WEKA Data Mining Software: An Update*, in *SIGKDD Explor. Newsl. June 2009, 11, 1*, 10-18p.

A SimplifiedCalo settings used in TAU analysis

```
export PERFORMANCE=1
export PHYSLIST=EmPhysics
export EMPROCESS=EmStandard

/run/verbose 1
/event/verbose 0
/tracking/verbose 0
/gun/particle e-
/gun/energy   50 GeV
/mydet/absorberMaterial Copper
/mydet/activeMaterial Scintillator
/mydet/isUnitInLambda 0
/mydet/absorberTotalLength 7000
/mydet/calorimeterRadius 3000
/mydet/activeLayerNumber 100
/mydet/readoutLayerNumber 20
/mydet/activeLayerSize 4.0
/mydet/update
/run/beamOn   500
```