

AATrack Review Report

Philippe Canal, Jim Kowalkowski

1 Introduction

1.1 Purpose

The purpose of this report is to list some of the problems we've encountered and possible changes needed in the AA package. This package requires these changes so that it can be better maintained and enhanced by others, so further performance improvements can be made on it, and so that it can be more easily studied in terms of code and effects of changing algorithm parameters.

We found that the AA package was quite difficult to change while trying to increase its execution speed. Two reasons for this are the difficulties we had in understanding the side effects of invoking functions within this monolithic package and lack of understanding how the quality of results shifts as a result of code changes. This report will highlight designs decisions that led to these difficulties and possible ways to remove some of the problems.

We recognize that producing a tracking package is a monumental effect and that the road-finding technique used in the AATrack package is known to yield good results. Due to time constraints, this report focuses almost entirely on negative aspects of the code within the package; this is, of course, its purpose. The author has included many constraints that reduce the number of combinations that need to be explored; effectively reducing the time it takes to locate tracks.

The document will suggest changes that will affect:

- Code readability,
- Code maintenance,
- The speed of the package,
- And validation (are the results correct).

1.2 Rationale

Why did we end up here? We chose an event from a high luminosity run that took long to process. After applying several patches to speed up the program and running the profiler, we made the following observations regarding timing of algorithms. The number on the left is the fraction of CPU time spent in this function and below.

```

GtrMetaPkg::processEvent(edm::Event&) : 0.80
  GtrHtfAAPkg::findTracks() : 0.74
    AA::HitMap::selectHypo(int&, bool) : .38
      AA::extrapolateTrack(...) : .29
        AA::TrkHypo::addHit(AA::HitFinder*) : .14
  GtrHtfAAPkg::addHtfTracks() : .22
  AA::findVrt(bool, bool) : .08
  AA::TrkBox::sectectTracks(book) : most of remaining time

```

One can draw several conclusions from this result. First, the tracking algorithms are taking up 80% of the CPU time on high luminosity events. We see that the HTF algorithm contributes about 22% of that time, leaving AA with 68%. This high number is the main reason we ended up discussing AA tracking.

Looking at the time spent in the top seven functions, we see that four of them are related to memory management (pthread functions are called from chunk functions – the memory management functions) and none contribute more than about 3% to the overall time.

CalNadaReco::runNADA(ErrorLog&, edm::Event&)	0.031
__pthread_alt_unlock	0.023
chunk_free	0.021
operator*(HepMatrix const&, HepSymMatrix const&)	0.020
__pthread_alt_lock	0.017
wash_list(vector<Trajphi>&, Hashmap<...> const&, bool)	0.016
chunk_alloc	0.016

We know that modern processors require good memory cache utilization to achieve high performance. If data structures are not designed with “locality of reference” in mind (putting things close together that are used together), program performance suffers as a result. Using too much caching or too many look up tables can have similar results. The observations that the memory management system is used somewhat heavily and the time is spread out so evenly amongst the functions lead us to some of the following conclusions:

- Data structures within algorithms are poorly designed or poorly organized,
- The data used by algorithms may not be efficiently accessed by the higher-level algorithms (an algorithmic strategy problem),
- And the choices of containers or collection classes used within the algorithms are suboptimal.

Another issue on modern processors is branching. One set of modification that made a difference was *removal of “if” statements from within loops*. This meant creating two or more separate loops that did one function instead of one loop performing many functions. This *change not only made the code easier to understand, but also sped it up*.

1.3 Complications

The available tools do not make it easy to tell if a function is invoked many times or if the code within a function takes a long time to run. What the tools are good for is allowing us to move up the call tree until we can account for significant amounts of time. Within the algorithms we targeted, mid-level algorithms or sub-algorithms cannot be identified for replacement (e.g. bubble sort, minimization routines, differential equation handling routines) without major upheaval.

It is difficult to predict what effect a change will have in execution speed before you make it and then run tests. Some of the changes that already took place such as the sin/cos and similarity transform improvements were first tested outside the algorithm and d0reco and then integrated after we had an estimate of the effect. Most of the items covered in this report may not be testable in this way. Some of the changes may not produce a substantial change in execution time and exist solely as a way to make the package more maintainable and more amenable to change and locating problems.

We had a difficult time evaluating the effects of making changes to the code. We have heard good things about recocert, but were unable to make use of it. Even if we could use it, it is our understanding that it is a high-level view and we were hoping for something which gave us more insight as to what and where the problems might lie (e.g. which tracks were missed or which fake tracks were accepted and why). We were also unable to get a MC event sample at the current luminosity to use for evaluating the results of the algorithm – again, this may be our fault for not making a properly formed request to the right people. One other thing that we could not locate was an MC event sample with data that looked like something we would see next year or the year after. We also have a difficult time starting up a graphical event display – this was mostly due to lack of locating the correct person to walk up through the procedure.

2 Overview

While making speed improvements to AA, we ran into many difficulties. In this section we discuss many of them at a high level and suggest ways to improve the situation.

2.1 Debugging

There are monitoring and debugging statements spread throughout the code. The facilities used have an impact on performance regardless if the debugging levels are set high enough to print messages or not. It would be useful to distinguish detailed trace information from information that is used to check if the algorithm is producing good or interesting results. The former should be able to be removed completely from the code if the proper compilation flags are set. The latter should be always present, but not used in heavily traversed paths of the algorithm. The output from this facility should be easy identified and selected from the output with any “grep” like command. A simple way to do this is to produce record like structure with a key at the first column, such as “AATrack”.

One way to track entrance and exit from routines is to use a technique like the one in file `AATrack/src/Waiter.hpp`. Here the *Waiter* class holds the data that needs manipulation upon entry and exit and a *WaiterOperator* class uses a constructor/destructor to do the manipulation. Preprocessor “`#ifdef`” statements can be used to nullify the effects of the class, which will cause its complete removal during compilation.

A simple technique for having code present during special build (for printing detailed variable contents) is to use assert-like macros. The purpose of the macro is to remove the code completely if the compile is an optimized one. For debug builds, the macros leave calls to monitoring facilities in.

Use of runtime assertions is also good practice. These can be used at strategic points in code to make sure that variables are what you expect them to be. `D0` prevents assertions from being removed from code even during an optimized build.

2.2 Global Variables

This library is littered with global variables. The entire package is driven off global variables. This feature makes it very difficult to know where the values of these variables are set, modified, and manipulated. The class structures give the appearance that more than one instance of any class can be constructed and used. The presence and use of global variables, in most cases, prevents this from happening. Many of the global variables are defined in `AA.hpp/AA.cpp` and are used and modified throughout the code.

A reconstruction class derived from the framework *Package* class is the place for variables with job-long life times. This is, of course, standard procedure for `D0`. A package object has a unique identity and state that lasts as long as the job. Using this technique allows for easier use of configuration information (comes in on the *Package* constructor. It appears that one of the reasons for the global variables is to allow the objects (and memory within) to live longer than one event. The *Package* constructor can use the *reserve()* method of some containers such as *vector* to preallocate work space. For other containers, such as *list* and *map*, we will need to check the implementation to see if a call to *clear()* deletes the memory and does not actually save the memory allocations. If deletions are done, then special allocators (the second hidden template parameter) must be used to make use of memory pools.

Another way to contend with the global variable situation is to remove them; just make the instances at the beginning of the event processing. The clean up of residual state may be just as expensive as generation of a new instance. We can probably evaluate and compare the cost of both types of operations in an isolated test.

2.3 Configuration from Arrays

Several of the classes are configured by reading arrays of doubles. One place this can be found is in the geometry. It is difficult to find the source of these numbers and what the meanings of the numbers at each index value are. What

makes matters worse is that the interpretation of values later in the array may depend on values earlier in the array.

The RCP system provides a simple to use interface that makes it easy to identify the names of each configuration parameter.

Context dependent interpretations of values in arrays should be removed completely. Each context should be given a complete set of numbers – unique for that context, where each value is named appropriately.

Does it make sense to be able to arbitrarily inject new configuration data arrays into an object at any time? This may be useful during algorithm testing, but it seems then that just constructing a new instance with the new configuration will do the job. Is it possible to remove the methods for reconfiguration? The idea of changing configuration parameters in production sounds like big trouble.

2.4 Overloaded meaning of classes

The interpretation of private data in some classes depends on values within that data. Methods of some classes are usable (and only make sense) depending on values in the private data. These classes can be thought of as having a multiple personality. **This is bad because code that uses these classes must ask what the current active “personality” is before the any methods of the objects are called.** Looking at the where the class is used does not give one a good idea of how the code is using it. A good example of this feature is in the *Elm* class, with an interface that includes all types of subdetectors.

One way to eliminate this feature is to create one class for each type of thing represented. In most cases code needs to work with specific objects anyway, and having the specific type makes it more obvious what kind of processing is going on. If all the various types are related by some common interface, or can be nested inside each other, then an abstract base class is in order.

2.5 Redundant code

There are several methods of some classes that contain nearly identical code and algorithms. Modifying one without modifying another is most likely hazardous. It is difficult to know of the existence of these multiple, similar algorithms without a close examination of many different parts of the code. One example of this is *State5::propagate*. The C++ template facility exists to address problems such as this.

Is redundancy ever appropriate? Yes, the changes we made to *HitMap::makeTrack* discussed in the postmortem section of this document introduce some redundancy. At the same time they increase readability and speed of execution. In many cases small, inline function can take the place of redundant code; the maintenance is in one place and the compiler replicates it for you.

2.6 Use of abstract classes

Abstractions such as the generic *Hit* exist almost solely for hits to be held in a common container. The problem with this is that the algorithm almost always wants to work with the concrete types of hits. Nearly all code using hits in the abstract form needs first to ask the hit what type it really is, cast it to that real type, and then run code specific to that type. In some cases the assumption is made that because we are in this body of code, the type must be this particular kind of *Hit* (e.g. *SMTHit*). The existing code does not use the `dynamic_cast` feature of C++, so if the assumption is wrong, the cost is corrupted results and memory. This feature affects the ability to make changes to this code. If the older code assumes a specific ordering of *Hit* objects and uses a hard cast to convert these object to there underlying type, then the object providing the ordered list is rigidly bound to the calling code. Bad side effects caused by upgrading parts of the algorithm may go unnoticed by the testing suite or my cause errors that are very difficult to trace down to the original source.

If the casts are converted to true `dynamic_casts`, then there will be a performance penalty. Mixing of objects in a container using this generic type of storage can also be inefficient because the code using the *Hit* needs to always first ask the actual flavor of the hit before doing the cast to the correct type.

An abstract or base class that knows about its derived types is not really serving the purpose that these constructs are made for. The main purpose of base classes is so derived types can be used in a general fashion using the base class interface. In many cases here, the base class is strictly used to mix types in a container.

One possible solution to this problem is to store each type is a separate container. Using a container of objects should be explored instead of using containers of pointers. Where and when pointers to objects make more sense, buffer pools can be used to reduce the burden on the memory management system. When code does need to make use of the type in abstract form, a container with pointers to abstract types can be used.

2.7 Overuse of lists

This package uses lists heavily to manage tracks and other information. Lists do not have good memory management characteristics for running in algorithms such as this. They also do not provide good locality of reference. In a few places lists are sorted many, many times. This sorting actually shows up in the performance profile. Sorting lists is an inefficient operation.

Unfortunately there is not a simple way to correct this situation. Here is a list of some of the things that have **worked well to improve performance in the past**:

- Changing from a list to a deque,
- Using a vector, sorting it when appropriate so that undesirable entries land at the end, the chomping the bad entries off the end (this technique works well with pointers to objects),

- Watching for unordered entries as new objects are added to the end of the list, and only sorting if it is necessary,
- Use a specialized allocator for the list to reduce memory management interactions.

2.8 Overuse of maps

Maps are very easy to use, but are not best for many situations. The lookup time is $O(n \log n)$ and the memory requirements are high compared to binary search in a vector. Iteration through a map is more costly than a sorted vector because the iterators traverse a tree-like structure. A map used as a sorted array is also likely to have poor locality of reference. If the map is used strictly for keeping an always-sorted data structure, then there might be a better way to solve the problem.

If a data structure is filled early on and traversed or searched many times, then a sorted vector is probably going to yield better performance than a map. If a data structure is used mostly for searches and ordered traversal is not important, then a hash table is probably better suited. Other projects have used a vector-like object that watches for items pushed on the back of the vector to be inserted out of order and marks them as such. The first access to an element, where the unordered flag is set, will cause the sort to occur.

2.9 Use of maps with double as key

Use of doubles as keys in a map/multimap does not make any sense because of the uncertainty in calculation results. On the Intel platform where 80 bit floating point arithmetic is done inside the processor and only 64 bits are represented in memory can lead to ambiguous results when comparing doubles for insertion or query into/from the map. Turning the key from a floating point value to a fixed point integer representation does resolve the problem within the map class, but still does not address the comparison problem; it is just pushed into a different piece of code.

2.10 Perplexing geometry

All different types of geometric or detector elements are fused into one class. Interpreting the indexing scheme is difficult and the storage and manipulation of these is inefficient due in part because of the base 10 encoding. Classes are not used to distinguish different types of detector components. The questions that can be asked of geometry are limited and to some extent produce pointers to internal data or parts of internal data. This feature is not in itself bad; it only becomes a problem when users of the code rely on internal details that should not be exposed. The topic of geometry is covered in more details later in this document.

2.11 State5

More utility classes and abstractions are necessary to make this easier to comprehend and modify. The data within the class can be made easier to understand and manipulate by introducing objects like Helix, and

CovarianceMatrix, along with a set of functions that operate on these objects. The interface of this class appears to perform many functions, including propagation through a non-uniform magnetic field, filtering, momentum and position calculations, smoothing, packing/unpacking, and internals resetting. In addition, many of the methods, such as *propagate* take arguments (as doubles and vectors) that are out parameters. The variable names did not give us much of a clue as to what the returned data was and how it is used.

A big problem with simple **out parameter** use is that is it not always clear when one is looking at the calling function, whether or not arguments past into the function will be modified or not. This is especially true when looking at an intermediate-level function; in other words function B in the call sequence A->B->C. The method *State5::findField(Hit* ph, double& li, Vector3& xg, double& cz, double& cr)* is an example. Here *ph* is not const and is used in further downstream calls. Without careful examination of the downstream call function signatures, it is not clear if parts of *ph* are actually modified as a result of being used as arguments.

Out parameters are, in many cases, useful. A good way to make use of out parameters is to make sure that the input parameters are const, and group all the output parameters into a small *struct*, that is filled out by the called method. The called method should be const, to easily indicate that the state of the object filling the struct has not be modified as a result of the call.

The method *State5::propagate(double l2, double& cz2, double& cr2)* appears like many of the other methods that, given a set of parameters, will return other information without being declared const. This method actually causes a side effect of altering the object state. Apparently this has caught someone's attention and a comment was added to make this more easily known.

2.12 Functions with side effects

Many of the functions perturb object state and global state when they are called. Making changes to callers of these functions is difficult, especially when these are called within loops. An easy-to-locate set of methods that fall into this category are the methods that return nothing and take no arguments. Examples are *void Elm::fillBorders()* and *HitMap::makeHits2D()*. Some return bools, such as *HitMap::newLayer3()* and *HitMap::newPoint2()*.

This method of triggering state changes works well with iterators. One reason for this is that an iterator has a single function – to advance its way through a data structure. Even the call (function name) helps express the action that is taking place (e.g. operator ++). It is not clear that the programming technique used in *HitMap* with all the new/next methods matches well with the iterator model; there are too many cached variables and dependencies in perturbations. Is also differs because the container also houses the cursor or position we are at instead of a separate object.

The method *TrkBox::selectTracks(bool)* illustrates another type of problem we had in analyzing the code. Here the code in this method goes out to global data

(*HitBox* in this case) and changes its state as a side effect of getting called. Seeing this has lead us to believe that this is a brittle system; changing the call ordering of any part of the system can cause unknown changes in results.

Another interesting example is *void Vrt::makeHit()*. Here *Vrt* object state is modified and so is the state of a global variable. Another example of side-effects was *State5::Propagate*, given in the *State5* section.

Methods that modify global state as a side effect of being called can be located by searching for variables used that start with "AA::".

2.13 Inconsistent use of const

Many methods do not modify the state of an object and are not declared *const*. A simple example is *State5::propagateDefault(double& c2, double& cr)*. This method obviously does not modify state and is not declared as *const*. Knowing if a method is truly *const* helps other developers better estimate the impact a change in the function will have.

2.14 Hard-coded constants

There are numerous unidentified constants within the code. It is difficult to know if some of the checks involving these constants are necessary or redundant. The constants should have a meaningful label associated with them or be configurable through RCP system if they can change. This will allow others to understand the purpose of the number and allow one place where changes can be made to that number.

2.15 D0 infrastructure usage

The AA package appears to avoid use of D0 infrastructure such as *ErrorLogger*.

Some facilities seem not to be used as intended, such as RCP. The code that manipulates RCPs is wrapped in exception catches to allow for default values in code. This type of coding is not allowed in algorithms at D0. Numerous examples can be found in file *AATrack/fw/AAPkg.cpp*.

The geometry constants appear to be pulled out of the D0 geometry objects using reasonably named methods and placed into arrays of doubles or floats. Within *AATrack* we only see the index into the double array, which makes it difficult to know what the value represents.

The interface to the AA package is a set of global functions and global data.

2.16 Little or no encapsulation

Some of the objects give out pointers to data members to clients to be directly manipulated or allow any object to adjust their member data directly. This makes it very difficult to know how and when the state of an object is affected. In addition, some of the client code expects a certain ordering of data in collection held within other objects. This prevents reorganization – both bodies of code have intimate knowledge of each others contents and are, in a way, locked together. An example of producing data members is

```

170     int q() const {return _q;}
171     Jet* jet() const {return _pj;}
172     int jetSign() const {return _impSign;}
173     PType pType() const {return _type;}
174     MuoGlb* muon() const {return _pmu;}
175     Trk* track() const {return _ptr;}

```

from class *Ptl*. Here even a const version of a Ptl can give out an unprotected copy of data members. Other examples are in the *ElmBox* methods *rLayers()* and *zLayers()*, and also *TrkBox::tracks()*.

There are many cases where structs are the correct solution. There is no encapsulation with structs; they are just data organizational components. It is the object that is a mix of struct and class that caused us problems because it was unclear where the true management of the data lived – or if the data was manipulated in many places.

Here is an example from *TrkHypo::addHit()*.

```

496     if(pit->missAxial()) {
497         _missIn.push_back(make_pair(pit->elementAxial(),AA::AXIAL));
498     } else if(pit->missStereo()) {
499         _missIn.push_back(make_pair(pit->elementStereo(),AA::STEREO));
500     }
501 }

```

Here the variable *pit* is of type *HitFinder*. In this case *pit* is used mostly as a bunch of state – you check a flag and then use other data in the object depending on the state. Another way to approach this problem could be to have a method called *HitFinder::addMisses(list_to_fill)* that performs this action as part of the *HitFinder* code instead of as part of the *TrkHypo* class.

2.17 Cached Variables

Classes like *HitFinder*, *HitMap*, and *State5* cache many variables. In some cases there variables form groups with common structure, but there is no *struct* which holds the information. The *HitMap* class is one example of this. Three positions are maintained using many individually named data members. A simple utility structure could help one in understanding this code and also allow for a single place to name and manipulate these variables.

In some cases, such as *MapC*, the interface actually forces a client class to cache several variables in order to manipulate this data structure. The dual iterator interface of methods *forward()* and *backward()* are examples of this.

These cached variables should be reviewed to see if they are really necessary. They increase the size of the object and also require code in many places to check if they are valid or need to be reset.

2.18 Derivation from Standard Containers

Example of classes derived directly from standard containers are *Chain*, *MapC*, *Trk*, and *Ptl*. These containers were not designed to be used in an inheritance tree and it is considered bad practice to do so. For one, there is no virtual destructor, so the derived object cannot be used as a base object in a reasonable fashion. The reason for this appears to be that several of the

methods of a standard container were needed in the derived class and that this was an easy way to grow the methods (implementation inheritance). This mostly just makes it difficult for someone to understand what methods are really necessary and useful in manipulating the derived object. Are methods splice, unique, merge, swap, and remove really appropriate for a *Chain*? A better way to organize this is to have the container as a private data member and produce methods that call through to the container methods. Another way is to use private inheritance and bring the base class methods forward into the derived with the *using* statement.

Another interesting case is the *Ptl* derivation from *Trk*. The *Trk* object does not appear to be designed for inheritance.

2.19 Important Call Sequences

Many of the objects require calls to methods in a particular sequence. The knowledge of this sequence is kept in code that is using the class. Changing the behavior of this class is made difficult because all the clients must be found and upgraded to reflex the different call ordering. Look at *AA:findTracks()* as a simple example of this behavior. The first two lines force the global hitMap object to makeHits2D() and then do a fill operation. Later the selectHypo() method is called which changes the global trkBox, followed by a trkBox.selectTracks, which first goes and modifies the global hitBox.

2.20 Variable Naming

It took us a long time to discover the meaning of some of the short variables, such as li, vxx, cr, xq, xl, xgs, _ha, _mz, _saa, etc.. More descriptive names or a translation table at the beginning of the function or file would have made our life much easier. The three character class names could also be made more descriptive.

3 Some Proposed Change Details

Here we present some ideas of how a few components can be improved.

3.1 Geometry

Each unique detector component should have a corresponding class that models it; this is idea behind object oriented programming. This means that the Elm and ElmBox classes and associated utility functions local(), index(), indexAxial(), indexStereo(), and id() need to be redesigned. For example, Elm can be broken up in at least four classes, one for CFT, SMT, SMTD, and FDISK. Each of the types of indices should have unique classes for easy manipulation and decoding. Base 10 index packing is very inefficient and should be converted to base 16 so that shifts and logical instructions can be used instead of multiplication and division.

A significant fraction of the code currently in AATrack is used to answer questions like 'what are the next detector elements in the direction followed by the current track?' and 'what are the coordinates and errors matrices in the local

frame of references?’ We strongly recommend that existing packages (Geant4, ROOT Geometry Package) which are designed to answer those questions be looked into as a replacement for the current geometry classes. At the very least those packages might provide a good template for redesigning the class structure and the objects relationship for the AATrack geometry.

3.2 The MapC Class

This utility class is one of the simplest in the system and exists somewhat in isolation of the other classes. It has several interface design defects that are worth mentioning. First, it is derived from a standard container, which was covered earlier in this report. The interface separates things that should travel as a group and exposes the pieces directly to the user. Three important pieces of information are phi, the start iterator, and the end iterator. The user is required to hold onto all three of these and pass them back to methods such as *forward()* and *nextBestC()* as arguments. If the iterators point at the end of the map, then they are treated in a special way. One way to improve this interface is to produce a special utility class, let’s call it *MapC::Cursor*, which holds all the information together as a unit. This class can be constructed with a phi and hold and initialize the iterators properly for use with MapC. This way the user is ensured that the data that is manipulated in subsequence calls is managed appropriately. It also makes the interface easier to understand and use properly.

3.3 Trk, TrkHypo, and Chain Notes

The main data structures within these classes are standard library linked lists, which are used to build up a graph or tree-like structure. Linked lists from the STL provide great flexibility and are not intrusive. As mentioned earlier, the properties of these containers may not be good for tracking. The “zipping” of tracks is apparently necessary because of the memory requirements of the tracking structures. Using alternative means of storing and manipulating tracks may reduce or remove the need for the zipping and improve algorithm speed. The *boost Graph library* uses one such alternative method for representing graphs and trees. It uses a series of simple arrays to represent vertices and edges in the graph. Another method is to use intrusive lists. In other words, include the pointers to parent, child, and sibling nodes directly in the nodes. Unfortunately both of these methods could end up not improving the readability of the code. It may be possible to use elements of these different techniques, such generating a simple vector of all considered tracks and using an index into that array within the track hypothesis tree data structure.

Tracks contain hits and it looks as though Hits point back to tracks (TrkZips). This circular dependency complicates the management of state in the program and makes it difficult to follow the logic. The Vrt and Trk classes have a relationship like this also. A Trk is a ChainList and a Chain point to a Trk. These relationships can almost always be represented as separate associations, i.e. tuples that tie the elements together.

The track parameters are represented in different ways in different parts of the code. In ChainZip, they are simply five floats. In State5, they are an array of doubles. There should be a single class that represents track parameters. If both double and float representations are needed, then this is an ideal candidate for a template class.

3.4 *HitFinder*

This class has a very confusing interface that manipulates, checks, and accesses many things in the package. We did not have time to understand how this class functions in enough detail to make good comments about it. A better geometry system, better separation of concerns, and better utility classes may allow this class to be simplified.

3.5 *Hit Classes*

One of the major hindrances to performance and algorithmic improvement in the current AATrack code is the organization of the hit processing code. The implementation of the tracking inside the CFT and SMT, which have a very similar general control flow but with sometime quite different details, are totally intermixed. For example, it is only after refactoring code similar to

```
    For each hit in 1st detector element
    ...
    if (areWeInCft) { ... }
    For each hit in 2nd detector element
    ...
    if (areWeInCft) { ... }
    For each hit in 3rd detector element
    ...
    if (areWeInCft) { ... }
into
if (areWeInCFT)
    For each hit in 1st detector element
    ...
    For each hit in 2nd detector element
    ...
    For each hit in 3rd detector element
    ...
Else
    For each hit in 1st detector element
    ...
    For each hit in 2nd detector element
    ...
    For each hit in 3rd detector element
    ...
    ...
```

that were able to detect some calculations that were totally useless in one case or another and some that were redundantly made in the inner most loops. Similar types of refactoring should be done in several places in the code (including the *HitFinder* class).

For example, we could have a *HitContainer* abstract class with two implementations: *HitSMTContainer* and *HitCFTContainer*. These classes would encapsulate the selection criteria for each of the detector. This arrangement would allow for a better separation and understanding of the selection algorithms.

The *HitContainer* should be organized in a similar fashion as the geometrical object. Hence the main driving routine might be

```
Given a track stub t
elemList = GeometryPackage->GetListOfElementsNextOnTrackPath(t,...)
for each element e in elemList
    find the corresponding HitContainer h
        for ( hitIter = h->GetBestHitsToExtendTrack(t);
            hitIter != end; ++hitIter) {
            if ( t->TryToExtendWithHit( *hitIter ) ) break;
        }
}
```

4 Speed Improvement Postmortem

In this section we discuss some of the improvements we have already made to the existing tracking code.

One of the first observations we made from the profiling data was that three of the top four functions, where the most time was spent, were **sin, cos and atan2**, with respective d0reco times of 4.5%, 4.3% and 3%. Since these functions are quite fundamental, these numbers really indicate that the functions are being called an extremely high number of times. However, we were still able to improve the performance by trading of some of the feature of the C library function and calling directly the Pentium trigonometric assembly instructions.

The C library function for sin and cos contain additional code to normalize and check the validity of the input and output. Since the inputs used in AATrack are already known to be valid, we could bypass these checks. In addition, in most cases both the sin and the cos of an angle were needed at the same time. In this case we took advantage of the Pentium instruction that does both calculations at once. We were also able to factor out some of the redundant calls after these changes were made.

The C library function for atan2 also contains additional code to set the value of errno in case of arithmetic error. Since AATrack assumes that the result of atan2 will be valid, we were able to bypass the error checking code and directly call the assembly instrument calculating atan2.

These two optimizations represented a significant gain in performance but should only be used as a last resort. The longer term solution would have been to reduce the number of calls to cos, sin and atan2 needed. This is currently quite difficult to do for AATrack due in part to the readability and maintenance issues listed in the previous sections.

Another straightforward gain was made by modifying a debugging function that was taking **a std::string object by reference and changing it const char***. In general, it is sufficient for performance reasons to pass objects by reference instead of by value. However, in this case the function was being called relatively often and was called exclusively by passing it a *const char** argument. This meant that despite the string reference, which was used to prevent an object copy, there were still a lot of temporary *std::string* objects generated (one per call). In addition, those constructed strings were useless in the common case

(running without debug message) since the function was not even using the parameter! The profiler helped us to quickly locate this problem.

The `std::list` method for calculating the distance between two elements was ranked 6th for the amount of time spent in itself. We traced down the calls as coming from `calls to std::list::size, which has to scan through the whole list`. However, in most cases this operation was not called to know the current number of element in the list but was called to know whether the list was empty or not. Replacing those calls by calls to `std::list::empty`, which only has to make one test, significantly improved performance.

Several of the functions in the top 20 most consuming functions were related to calculating the `similarity transform of 5x5 matrices`. After some investigations we realized that the implementation being used was inefficient. Part of the inefficiency was due to unnecessary memory copy and others were due to the calling of a virtual member function in the inner most loop of the calculation. In addition, the return value of this member function was not dependent on any of the loops indices! We then proceeded to test a custom version of the similarity transform calculation. This new implementation was C++ template based, to execute has much of the calculation as possible at run-time and to allow the compiler more opportunity to optimize the code and was designed to work for 5x5 matrices. The result was an amazing 10 fold improvement in performance, reducing the runtime for d0reco by 4%!

The 10th most time consuming method was `floor()`. This function was being used in normalizing angle between 0 and 2 pi or between $-\pi$ and $+\pi$. After careful examination of the code we realized that in most cases the range of the input was known and that a custom normalization routine taking this information in consideration could be used. This allowed replacing, in most cases, a normalization that involved calls to `floor` and floating point multiplication and divisions (all slow operations) by just a if statement and a floating point addition or subtraction.

One of the issues we noted earlier was the need to replace maps and multimaps with more appropriate containers. As a simple example we decided to change the underlying container of the `class MapC from a multimap<double,Hit*> to a vector<pair<double,Hit*>`. We chose a vector of `pair<double,Hit*>` to minimize the changes needed. This new implementation only required the writing of a `lower_bound()`, `upper_bound()`, `find()`, and `insert` wrapper functions (all one or two liners). The next logical step would have been to sort this container only on demand instead of inserting the values in sorted order. However, this requires a careful analysis of the code using these `MapC` objects to find a proper place for a one time sort. In order for this step to be possible, the algorithm needs to be such that there is two separate phases: one filling the maps and one using the maps. However, the current AATrack seems to intermix these two phases.

AATrack also contains several cases where the following code pattern is used (directly or indirectly):

```

For each hit in 1st detector element
...
if (areWeInCft) { ... }
For each hit in 2nd detector element
...
if (areWeInCft) { ... }
    For each hit in 3rd detector element
        ...
        if (areWeInCft) { ... }

```

This pattern actually has two major disadvantages. Not only does it imply a lot of **redundant, time consuming, “if statements” inside the inner most loops**, but it also make it difficult to find unnecessary and redundant calculations. In the particular case of *AA::HitMap::makeTrack*, after doing the re-factoring we discovered that the code actually looked like:

```

For each hit in 1st detector element
...
if (areWeInCft) { ... }
For each hit in 2nd detector element
...
if (areWeInCft) { ... }
    For each hit in 3rd detector element
        ...
        ... (a) Calculation needed in only one of the detectors
        ... (b) Calculation only depends on the outer loop indices
        if (areWeInCft) { ... }

```

By separating *makeTrack* in 3 new functions (*makeTrackCF*, *makeTrackSMT* and *makeTrackSMTD*), we were able to save some computation time by completely removing the code (a) from detector’s routine that did not need it and to move the code (b) at the proper places in the loops.

5 Improvement Plan

Here is our opinion of what changes should take place in this package and in what order. All of the categories of changes listed here are covered in the previous sections. The order is based on our limited knowledge of tracking and the current code. The items further down in the list are considered more difficult.

- **Constants replacement**: Addresses code readability and may help locate redundant checking of constraints. This organization helps when values of constants need adjust and study.
- **Container replacements and addition of buffer pools**: This applies to containers that are readily replaced or can easily have a buffer pool attached to them. These changes should have a direct effect on speed by reducing memory allocations and moving items closer together.
- **Geometry modifications**: This should make the code easier to understand and simplify making algorithmic adjustments. These changes should also speed up the algorithm.
- **Hits and collections of Hits**: Changes here should have an effect on speed and also on the ability to comprehend and adjust this part of the system.
- **State5**: This involves separating out things like Kalman filter, position determination, and helix manipulations. These changes are targeted at

improving a persons understanding of the code so that performance improvements and adjusts can be made in the future.

- **Track related structures:** This includes things like Chains, Tracks, Vertices, and extrapolation. Some of the container improvements from previous steps may be difficult to implement in these parts of the code. This is the time to try and replace those containers. Here we want the classes to be examined to see if the problem addressed can be further decomposed (into helper or utility classes). The data structure improvements could increase the execution speed.

6 Conclusion

Other packages such as HTF also contain many of the problems listed in this document, especially concerning use of the standard containers. HTF **will likely also benefit from some of the techniques used to improve AATrack.**

One of the most important guidelines to follow when working in C++ on a subsystem such as this is to produce single-purpose objects that capture and control their state in a clear and easy to understand manner. Addressing the issues in this document with this goal in mind will help reduce the complexity of this code.

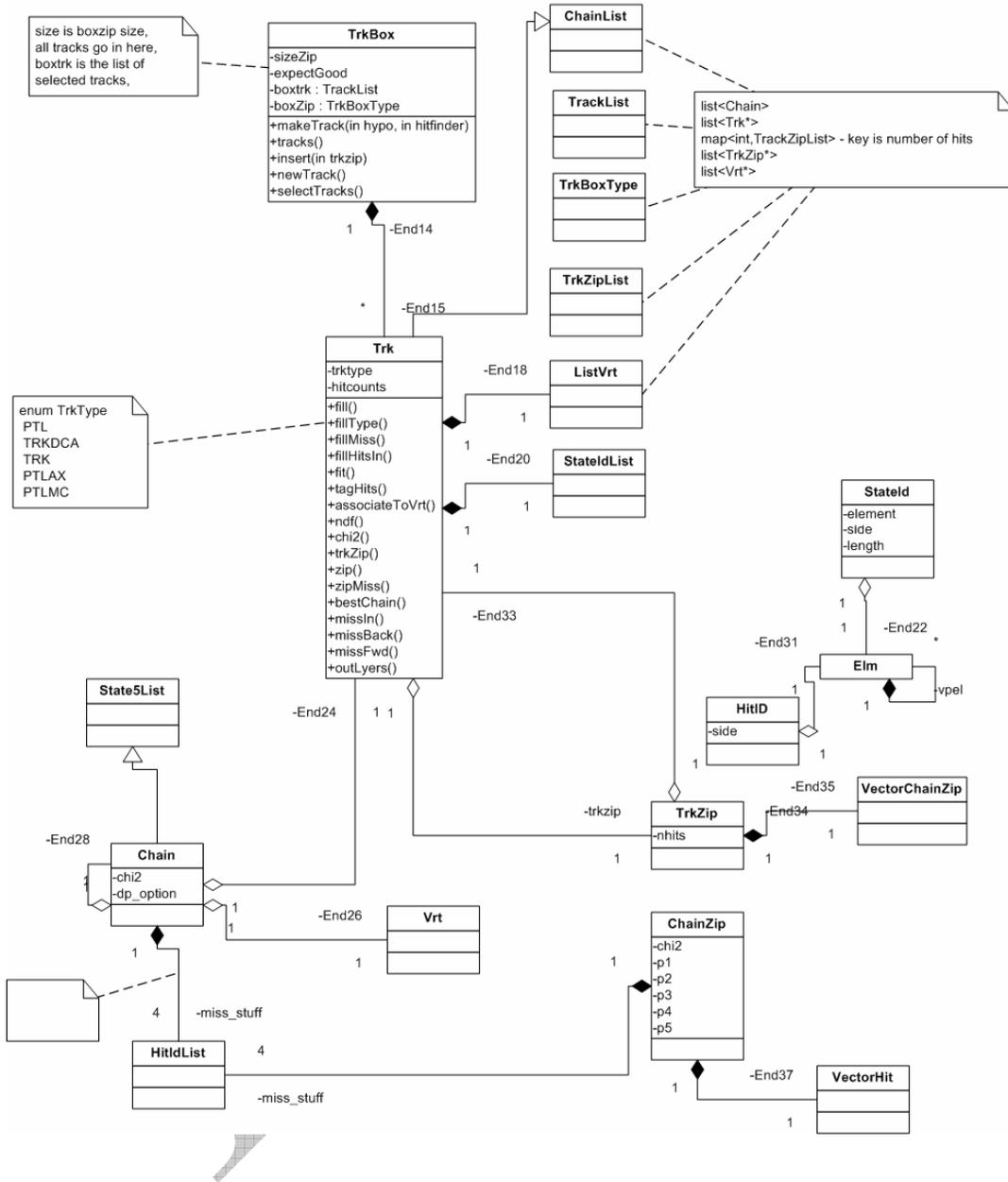
Following the **Improvement Plan** within this document and addressing the items listed in **Complications** will help to reduce the execution speed of this package and make it possible for others to understand and upgrade this body of code.

The presence of the **Improvement Plan** section indicates that we think that it is best to start with the current AATrack package and make a series of changes to it. **For many of the items in the improvement list, we expect the impact on this package to be dramatic. The algorithm concepts and rules will remain intact, but the implementation may change significantly.** Our hope is that sections can be modified, replaced or revamped one at a time so that performance changes can be observed.

7 APPENDIX

This section contains diagrams we drew in order to better understand the organization of the data structures within AATrack. In many cases we produced the bare minimum necessary to understand the relationships between the different classes. In nearly all the classes, the methods and data members are truncated (i.e. left out) and only the most relevant features exist. We also took shortcuts in producing the diagrams and rely on attached notes because of the complexity of working with UML and templates instantiations and typedefs.

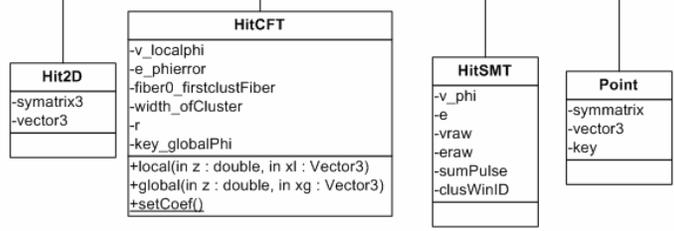
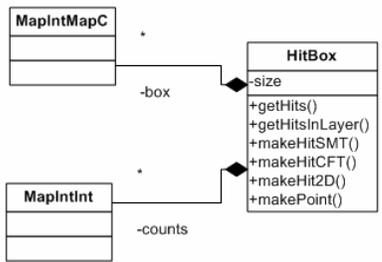
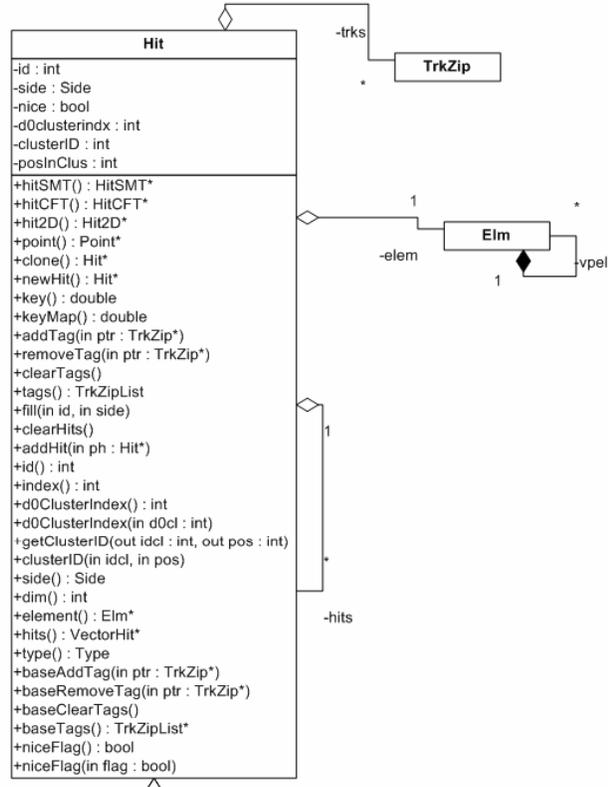
Tracks



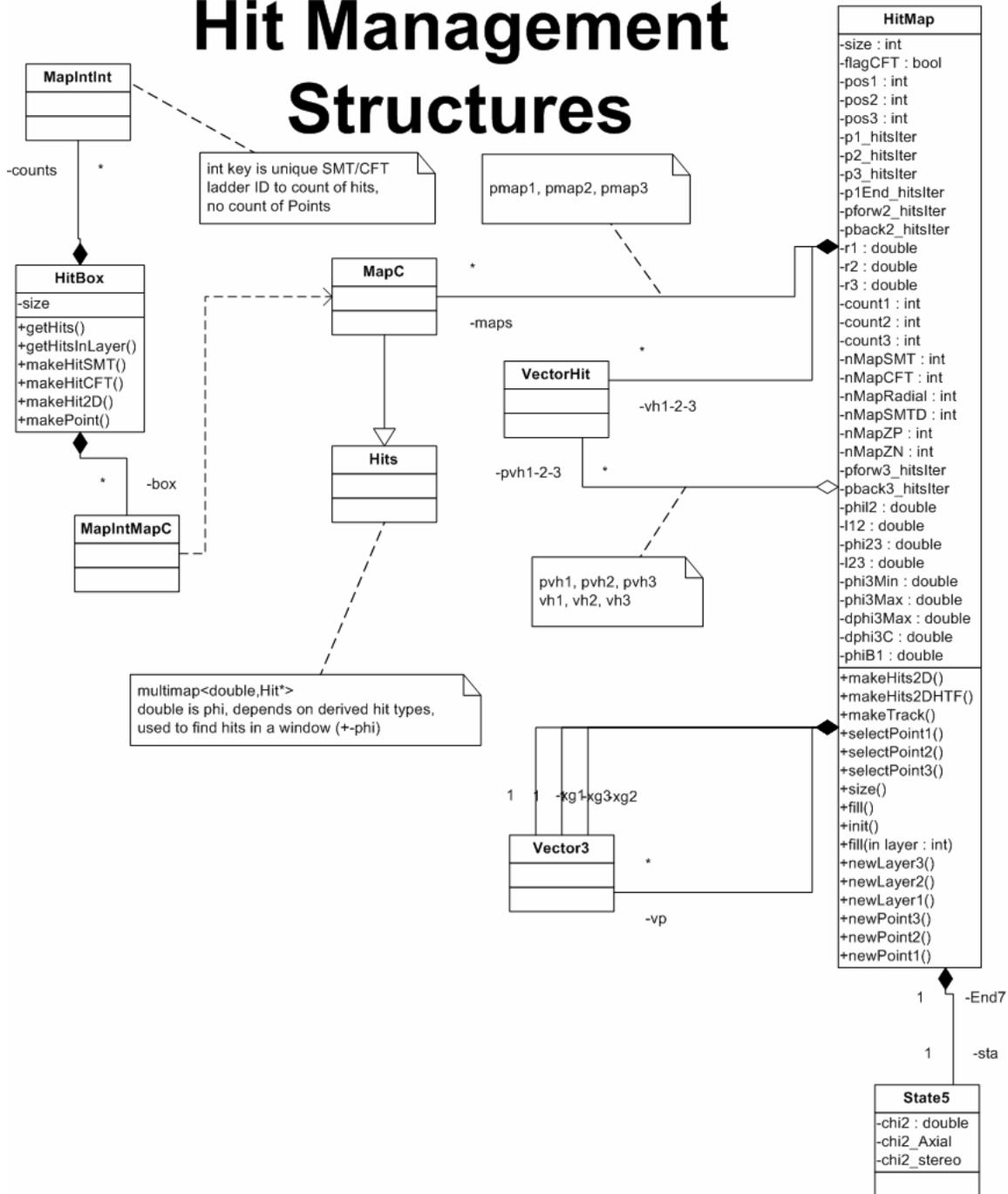
Hits

```

HitFinder
-forward : bool
-stop : bool
-flagCFT : bool
-checkAccepted : bool
-accepted : bool
-statAx : Stat
-statSt : Stat
-inax : bool
-inst : bool
-phitsAx : MapC*
-phitsSt : MapC*
-pelAx : Elm*
-pelSt : Elm*
-pst : State5*
-li
-r
-tl
-z0
-lnew
-lnewSt
-vax
-sax
-vst
-sst
-rend
-zend
-xg : Vector3
-indend : int
-points
-p
-miss
-checked
-plr
-pendr
-plz
-pendz
+HitFinder()
+selectPoint() : bool
+selectPoint(in found : bool) : bool
+selectPoint(in found : bool, in best : bool) : bool
+selectPoint(in state5_start, in state5_end, in best : bool) : bool
+hit() : Hit*
+getHits(inout hitlist_iter_start, inout hitlist_iter_end, inout count : int&)
+length() : double
+lengthStere() : double
+miss() : HitIdList
+checked() : HitIdList
+setFlagCFT(in flag : bool)
+missAxial() : bool
+missStere() : bool
+elementAxial() : Elm*
+elementStere() : Elm*
+statAx() : Stat
+statSt() : Stat
    
```



Hit Management Structures



Geometry

