Chrono & Stat Service

Vanya Belyaev[1]
LAPP/Annecy & ITEP/Moscow

August 3, 2004

Internal Note

Issue :	1
Revision :	2
Reference :	LHCb-COMP-Offline-2004-064
Created :	December 1, 1999
Last modified :	August 3, 2004

[1]E-mail:Ivan.Belyaev@itep.ru

Abstract

The detailed description of standard Chrono & Stat Service is presenetd. The Service is designed for simple code profiling and statistical monitoring.

Contents

1 Chrono & Stat Service and `IChronoStatSvc` Interface **3**

1.1 Access to the Chrono & Stat Service 3

1.2 **`IChronoStatSvc`** abstract interface 3

1.3 Code profiling 4

1.4 Statistical monitoring 6

2 `Chrono` and `Stat` helper classes **8**

2.1 **`Chrono`** 8

2.2 **`Stat`** 8

3 Properties of standard `ChronoStatSvc` service **9**

3.1 Properties for profiling 9

3.2 Properties for statistics 9

4 Performance consideration **10**

Chapter 1

Chrono & Stat Service and IChronoStatSvc Interface

Chrono & Stat Service is one of standard components directly visible by Algorithm object. This service is designed to fulfil two different task:

- chrono profiling of the codes (”Chrono” part)
- simple statistical monitoring of useful quantities (”Stat” part)

1.1 Access to the Chrono & Stat Service

Communication with Chrono & Stat Service is performed via an abstract interface **IChronoStatSvc**. Each Algorithm object[1] has the method for accessing to the pointer to an abstract interface of type *IChronoStatSvc*:

```
1  // The standard Chrono & Stat Service,
2  // Return a pointer to the service if present
3  IChronoStatSvc* chronoSvc() const;
```

For accessing the Chrono & Stat Service outside the Algorithm object, one should use standard method of service location, using either the name of the service *”ChronoStatSvc”* or the unique service ID: **extern const InterfaceID& IID_IChronoStatSvc**.

1.2 IChronoStatSvc abstract interface

The most important methods from abstract **IChronoStatSvc** interface are listed here:

```
1  class IChronoStatSvc : virtual public IService
2  {
3    ....
4  public:
5    ...
6    // the actual type for Chrono Tag
7    typedef    std::string      ChronoTag   ;
8    // the actual type for Stat   Tag
9    typedef    std::string      StatTag     ;
10   // type of the Flag   variable for statistics
```

[1] The method is also available through the base class `GaudiTool` for Tool object

```
11   typedef     double            StatFlag    ;
12   // type of the Weight variable for statistics
13   typedef     double            StatWeight  ;
14   // Type of the delta-time
15   typedef     longlong          ChronoTime  ;
16   ...
17   enum ChronoStatus
18     { UNKNOWN = 0 , RUNNING , STOPPED } ;
19   enum ChronoType
20     { USER    = 0 , KERNEL  , ELAPSED } ;
21   ...
22
23   virtual StatusCode    chronoStart
24           ( const ChronoTag& t ) = 0;
25   virtual StatusCode    chronoStop
26           ( const ChronoTag& t ) = 0;
27   virtual ChronoTime    chronoDelta
28           ( const ChronoTag& t ,
29             ChronoType       f ) = 0;
30   virtual StatusCode    chronoPrint
31           ( const ChronoTag& t ) = 0;
32   virtual ChronoStatus chronoStatus
33           ( const ChronoTag& t ) = 0;
34
35   virtual StatusCode    stat
36           ( const StatTag&    t ,
37             const StatFlag&   f ,
38             const StatWeight& w ) = 0;
39   virtual StatusCode    statPrint
40           ( const StatTag&    t ) = 0;
41
42
43   virtual const ChronoEntity* chrono
44           ( const ChronoTag& t ) const = 0 ;
45   virtual const StatEntity*   stat
46           ( const StatTag&   t ) const = 0 ;
47
48   ...
49 };
```

1.3 Code profiling

Profiling is performed by using **`chronoStart`** and **`chronoStop`** methods inside the codes to be profiled, e.g:

```
1  /// ...
2  IChronoStatSvc* svc = ...
3  ///  start
4  svc->startChrono( "Some Tag" );
5  /// here some user code are placed:
6  ...
7  /// stop
8  svc->stopChrono( "SomeTag" );
```

The profiling informations accumulates under the *tag* name, given as an argument for these method. The

Service performs the measurement of the time between subsequent calls of **`startChrono`** method and **`endChrono`** method with the same *tag*. The later is important, since from the sequence of calls

```
1 svc->endChrono("Tag");
2 svc->endChrono("Tag");
3 svc->startChrono("Tag");
4 svc->startChrono("Tag");
5 svc->endChrono("Tag");
6 svc->endChrono("Tag");
7 svc->startChrono("Tag");
8 svc->startChrono("Tag");
9 svc->endChrono("Tag");
```

only the elapsed time between 3 and 5 lines and the elapsed time between 7 and 9 lines would be accumulated.

The information could be printed either directly using **`printChrono`** method of standard Chrono & Stat Service, or could be printed in the final table of profiling information.

The detailed information is available through the helper class **`ChronoEntity`**.

```
1 // get the full information
2 const ChronoEntity* entity = svc->chrono("Tag");
```

The class **`ChronoEntity`**, defined in the file **`GaudiKernel/ChronoEntity.h`** is equipped with set of methods which allow detailed run-time inspection of profiling information. The major public methods are listed here:

```
1  class ChronoEntity
2  {
3    ...
4    public:
5    /// return the status of chrono
6    IChronoStatSvc::ChronoStatus      status() const ;
7     /// number of chrono measurements
8    unsigned long          nOfMeasurements    () const ;
9
10   /// minimal measurement for ``user'' time
11   long double            uMinimalTime       () const ;
12   /// minimal measurement for ``kernel'' time
13   long double            kMinimalTime       () const ;
14   /// minimal measurement for ``ellapsed'' time
15   long double            eMinimalTime       () const ;
16
17   /// maximal measurement for ``user'' time
18   long double            uMaximalTime       () const ;
19   /// maximal measurement for ``kernel'' time
20   long double            kMaximalTime       () const ;
21   /// maximal measurement for ``ellapsed'' time
22   long double            eMaximalTime       () const ;
23
24   /// total ``user'' time
25   long double            uTotalTime         () const ;
26   /// total ``Kernel'' time
27   long double            kTotalTime         () const ;
28   /// total ``Elapsed'' time
29   long double            eTotalTime         () const ;
30
31   /// total time (``user'' + ``kernel'')
32   long double            totalTime          () const ;
```

```
33
34   /// sum of squared ''user''   time intervals
35   long double           uStatistics       () const ;
36   /// sum of squared ''Kernel'' time intervals
37   long double           kStatistics       () const ;
38   /// sum of squared ''Elapsed'' time intervals
39   long double           eStatistics       () const ;
40
41   /// average ''User''    Time
42   long double           uMeanTime         () const ;
43   /// average ''Kernel'' Time
44   long double           kMeanTime         () const ;
45   /// average ''Elapsed''    Time
46   long double           eMeanTime         () const ;
47
48   /// r.m.s ''User'' Time
49   long double           uRMSTime          () const ;
50   /// r.m.s ''Kernel'' Time
51   long double           kRMSTime          () const ;
52   /// r.m.s ''Elapsed'' Time
53   long double           eRMSTime          () const ;
54
55   /// error in mean ''User''   time
56   long double           uMeanErrorTime    () const ;
57   /// error in mean ''Kernel'' time
58   long double           kMeanErrorTime    () const ;
59   /// error in mean ''Elapsed''   time
60   long double           eMeanErrorTime    () const ;
61
62   /// print the chrono;
63   std::string           outputUserTime    () const ;
64   /// print the chrono;
65   std::string           outputSystemTime  () const ;
66   /// print the chrono;
67   std::string           outputElapsedTime () const ;
68
69   ...
70 };
```

This class allows to inspect the total, minimal, maximal, mean, RMS and uncertainty in mean evaluation for “User”, “Kernel” and “Elapsed” time measurements. Also the helper statistics of accumulated corresponding squared time intervals is available, which allow to evaluate other statictical quantities e.g. differents statistical moments.

1.4 Statistical monitoring

Statistical monitoring is performed by using the **stat** method inside user code:

```
1 /// ... Flag and Weight to be accumulated:
2 svc->stat( " Number of Tracks " , Flag , Weight );
```

The statistical information contains the ”accumulated” *flag* , which is the sum of all `Flags` for the given tag, and the ”accumulated” *weight*, which is the product of all `Weights` for the given tag. The information is printed in the final table of statistics.

The detailed information is available through the helper class **StatEntity**.

```
// get the full information
const StatEntity* entity = svc->stat("Tag");
```

The major pubulic methods of class **StatEntity**, defined in the file **GaudiKernel/StatEntity.h** are listed here:

```
class StatEntity
{
  ...
  public:
  /// number of measurements
  const size_t         nEntries            () const ;

  /// minimal flag
  const StatFlag       flagMin             () const ;
  /// maximal flag
  const StatFlag       flagMax             () const ;
  /// accumulated "flag"
  const StatFlag       flag                () const ;
  /// accumulated "flag squared"
  const StatFlag       flag2               () const ;
  /// mean value of flag
  const StatFlag       flagMean            () const ;
  /// r.m.s of flag
  const StatFlag       flagRMS             () const ;
  /// error in mean value of flag
  const StatFlag       flagMeanErr         () const ;

  /// minimal "weight"
  const StatWeight     weightMin           () const ;
  /// maximal "weight"
  const StatWeight     weightMax           () const ;
  /// accumulated "weight"
  const StatWeight     weight              () const ;
  /// "mean harmonic" weight
  const StatWeight     weightHarmonicMean  () const ;

  /// output of StatEntity
  const std::string    stringOutPut        () const ;
  ...
};
```

This class allows to inspect the total, minimal, maximal, mean, RMS and uncertainty in mean evaluation for “Flag” variable. Also the helper statistics of accumulated squared “Flag” is available. For “Weight” variable the accumulated “Weight”, minimal and maximal values and the harmonic mean is available.

In some sense the profiling could be considered as statistical monitoring with variable `Flag` equals to elapsed time of the process.

Chapter 2

Chrono and Stat helper classes

To simplify the usage of Chrono & Stat Service, two helper classes were developed: **Chrono** and **Stat**. Using these utilities, one hides the communications with Chrono & Stat Service and provides more friendly environment.

2.1 Chrono

Chrono is a small helper class, which invokes **startChrono** method of Chrono & Stat Service in the constructor and invokes **endChrono** method in the destructor. It must be used as *automatic local object*. It performs the profiling of the codes since its own creation till the end of the current scope, e.g:

```
1 /// ...
2 { // begin of the scope
3    Chrono chrono( svc , "ChronoTag" ) ;
4    /// some codes:
5     ...
6    ///
7 } // end of the scope
8 /// ...
```

For usage of **Chrono** utility one must include the file **GaudiKernel/Chrono.h**.

If the Chrono & Stat Service is not accesible `svc=0`, the **Chrono** object does nothing.

2.2 Stat

Stat is a small helper class, which invokes **stat** method of Chrono & Stat Service in the constructor:

```
1 /// ...
2 Stat stat( svc , "StatTag" , Flag , Weight ) ;
3 /// ...
```

For usage of **Stat** utility one must include the file **GaudiKernel/Stat.h**

If the Chrono & Stat Service is not accesible `svc=0`, the **Stat** object does nothing.

Chapter 3

Properties of standard ChronoStatSvc service

3.1 Properties for profiling

Profiling properties of the standard Chrono & Stat Service which are configurable via *job options file* and their default values are:

```
// deside if the final printout should be performed;
ChronoStatSvc.ChronoPrintOutTable = true;
// if User   Time information to be printed?;
ChronoStatSvc.PrintUserTime     = true        ;
// if System Time information to be printed?;
ChronoStatSvc.PrintSystemTime   = false       ;
// define the destination of the table to be printed;
ChronoStatSvc.ChronoDestinationCout = false;
// print level for profiling  ( according to MSG::Level)
ChronoStatSvc.ChronoPrintLevel       = 3 ;
// if printout is to be performed,
// should one take care about some ordering?
ChronoStatSvc.ChronoTableToBeOrdered = true ;
```

Tables are ordered according the total elapsed time.

3.2 Properties for statistics

Statistical properties of the standard Chrono & Stat Service which are configurable via*job options file* and their default values are:

```
// deside if the final printout should be performed;
ChronoStatSvc.StatPrintOutTable = true;
// define the destination of the table to be printed;
ChronoStatSvc.StatDestinationCout = false;
// print level for profiling  ( according to MSG::Level)
ChronoStatSvc.StatPrintLevel       = 3 ;
// if printout is to be performed,
// should one take care about some ordering?
ChronoStatSvc.StatTableToBeOrdered = true ;
```

Chapter 4

Performance consideration

Implementation of Chrono & Stat Service used two **`std::map`** containers. Therefore for very frequent calls it could cause the performance penalty. Usually this penalty is negligible with respect to the elapsed time of algorithms, but it is worth to avoid both the direct usage of Chrono & Stat Service as well as the usage of it through **`Chrono`** or **`Stat`** utilities inside the internal loops:

```
1  /// ...
2  {  /// begin of the scope
3  Chrono chrono( svc , "Good Chrono");  /// OK
4  long double a = 0 ;
5  for(  long i = 0 ; i < 1000000 ; ++i )
6   {
7   Chrono chrono( svc , "Bad Chrono");   /// not OK
8   /// some code :
9     a += sin( cos( sin( cos( (long double) i ) ) ) );
10  ///  end of codes
11 Stat    stat  ( svc , "Bad Stat", a ); /// not OK
12  }
13 Stat    stat  ( svc , "Good Stat", a); /// OK
14 } ///  end of the scope!
15 /// ...
```