Identifying Memory Allocation Patterns in HEP Software

Sami Kama[1], Nathalie Rauschmayr[2]

[1]Southern Methodist University
[2]IT-DI-LCG-CERN

11-Oct-2016

Introduction

- HEP Applications usually consume large amounts of memory
- Cost of memory per core is increasing
- Significant amount of this memory is due to temporary allocations
- Memory layout and access patterns are keys to performance

Introduction

- HEP Applications usually consume large amounts of memory
- Cost of memory per core is increasing
- Significant amount of this memory is due to temporary allocations
- Memory layout and access patterns are keys to performance

$\Rightarrow$ We need to understand how and why we allocate/deallocate memory

FOM-Tools

FOM-Tools provides a means to analyze memory allocation and utilization (`https://gitlab.cern.ch/fom/FOM-tools.git`)

- Identify unused memory allocations by making use of kernel features
- Intercept memory allocations/deallocations and record
  - time
  - address
  - size
  - type of allocation
  - stack trace

Performance

- Runtime overhead is 5-6x depending on profiled application and tool configuration
- Memory overhead is negligable ($\ll 100\ MB$)
- Typical output size, depending on the job and tool configuration is O(10GB) for compressed, O(100GB) for uncompressed

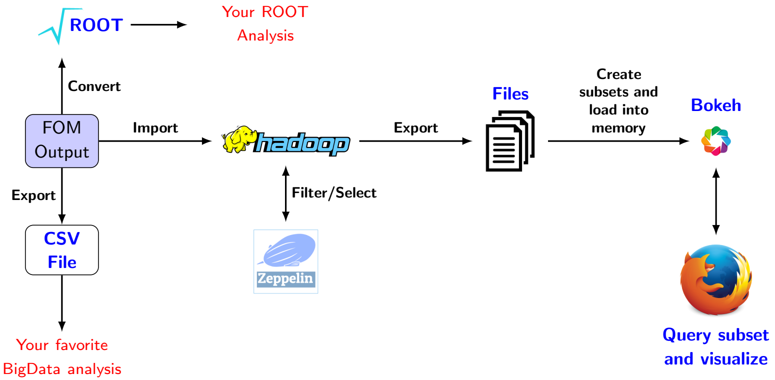
Key Metrics

Using data from FOM-Tools, we can calculate, globally or per class/stack

- Density: How many (de)-allocations per time unit
- Variation: How do consecutive allocations differ in size
- Locality: How close are allocations in address
- Contention: How many concurrent memory allocations
- **Lifetime: How long does allocated memory live**

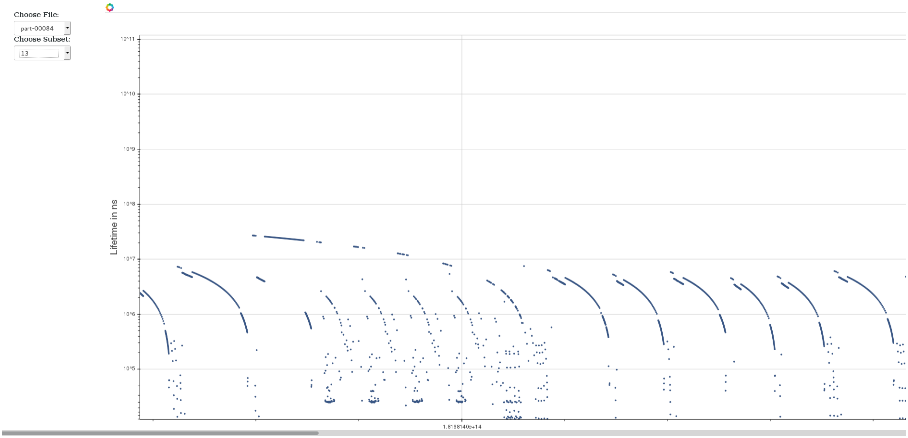
Data Analysis

FOMTools output can be analyzed in various ways



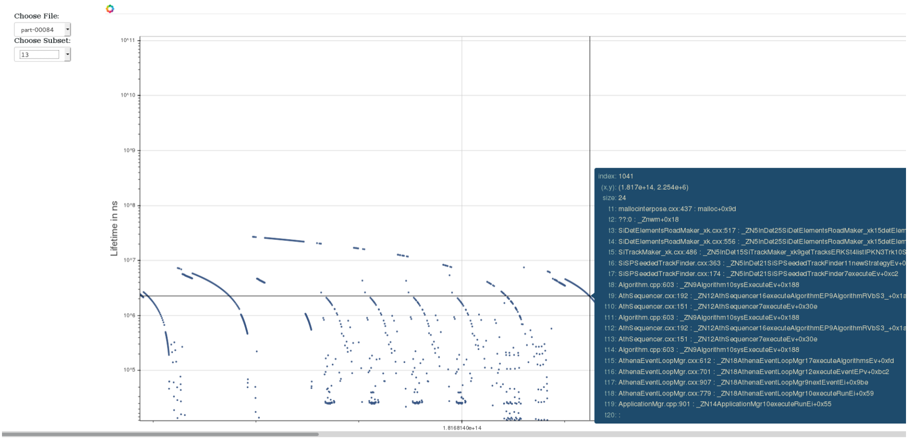
Visualization

Results are visualized on browser with interactive bokeh plots



Visualization

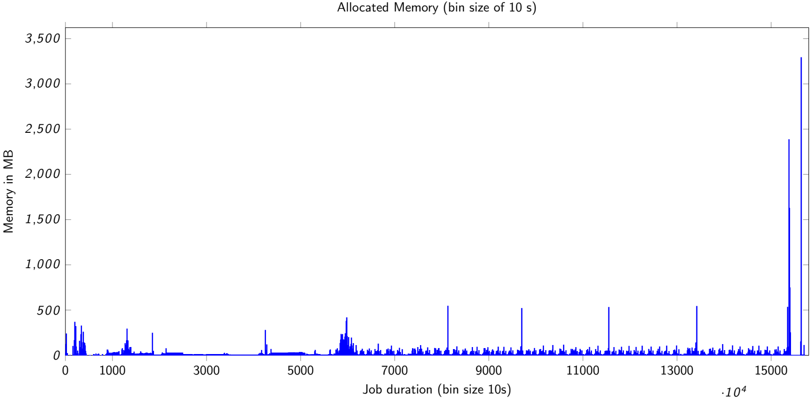
Tooltips give stack-trace for the given point



Example-1: General Exploration

A Reconstruction job with 50 events:

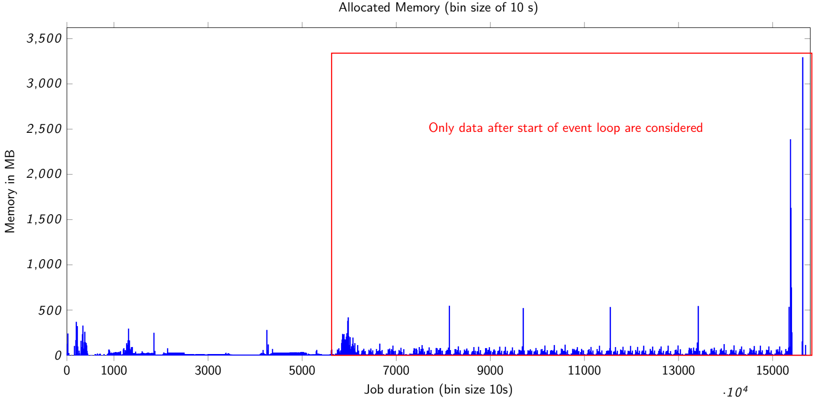
- Generated roughly 870M records
- About 90 GB uncompressed binary output file (with 20 deep stack traces)



Example-1: General Exploration

A Reconstruction job with 50 events:

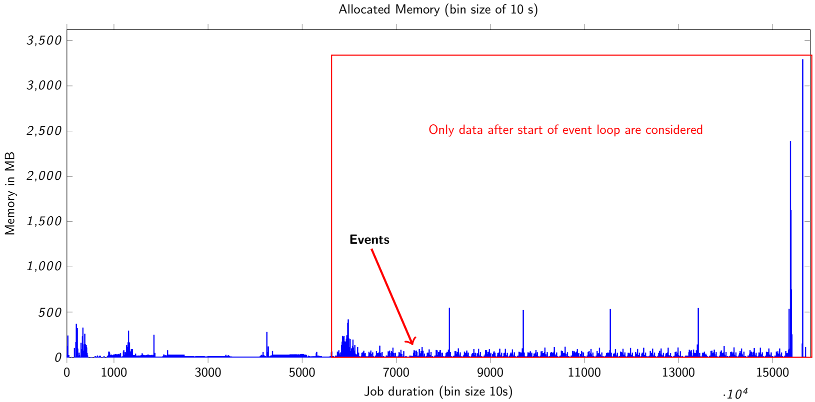
- Generated roughly 870M records
- About 90 GB uncompressed binary output file (with 20 deep stack traces)



Example-1: General Exploration

A Reconstruction job with 50 events:

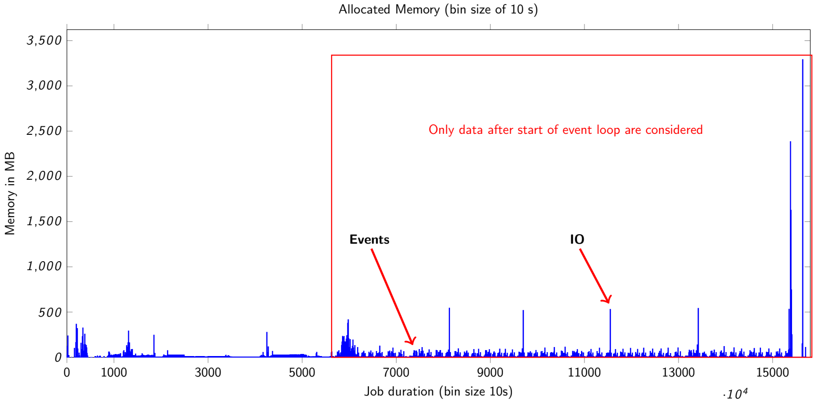
- Generated roughly 870M records
- About 90 GB uncompressed binary output file (with 20 deep stack traces)



Example-1: General Exploration

A Reconstruction job with 50 events:

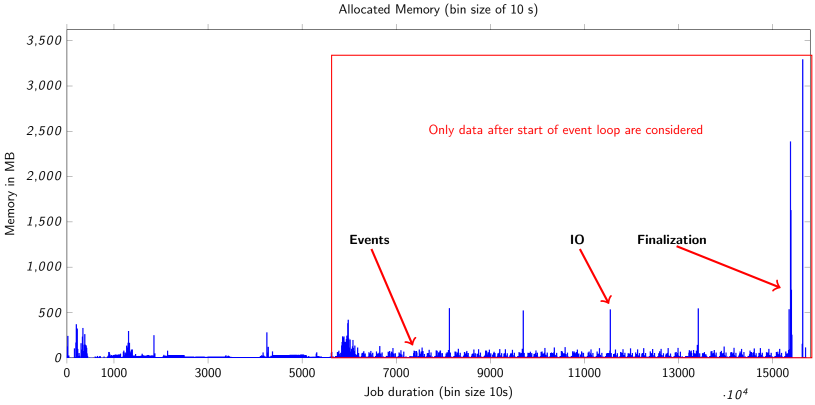
- Generated roughly 870M records
- About 90 GB uncompressed binary output file (with 20 deep stack traces)



Example-1: General Exploration

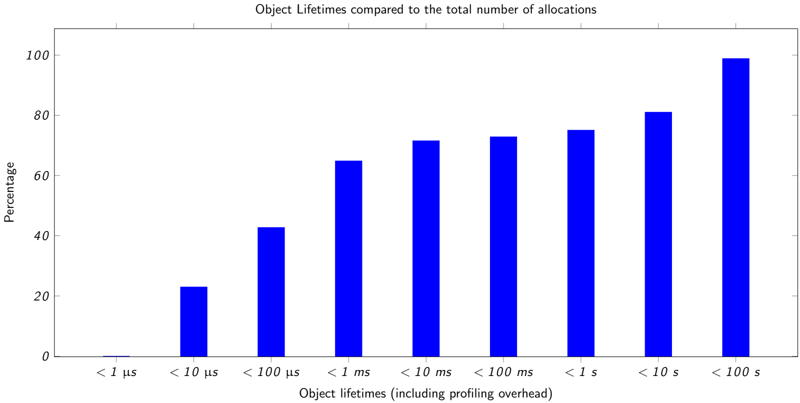
A Reconstruction job with 50 events:

- Generated roughly 870M records
- About 90 GB uncompressed binary output file (with 20 deep stack traces)



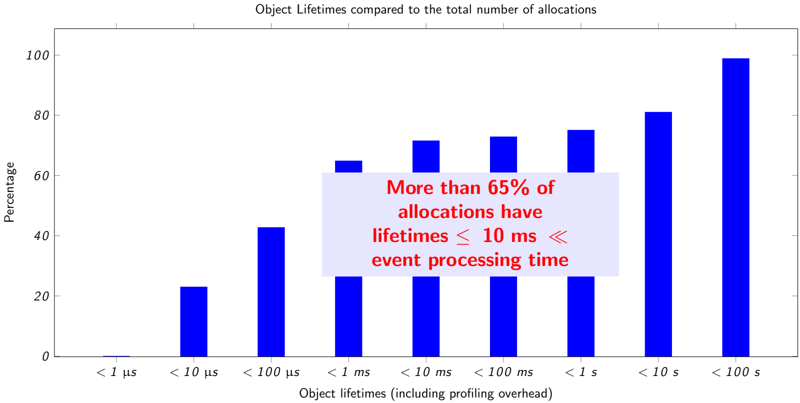
Object Lifetimes

Looking at the lifetime of allocations



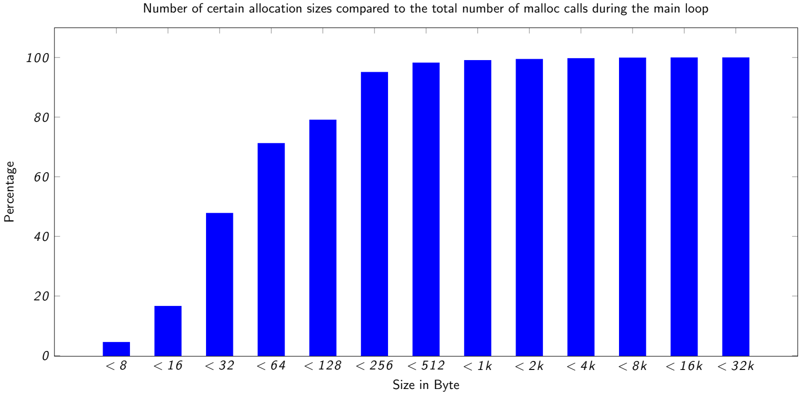
Object Lifetimes

Looking at the lifetime of allocations



Object Sizes

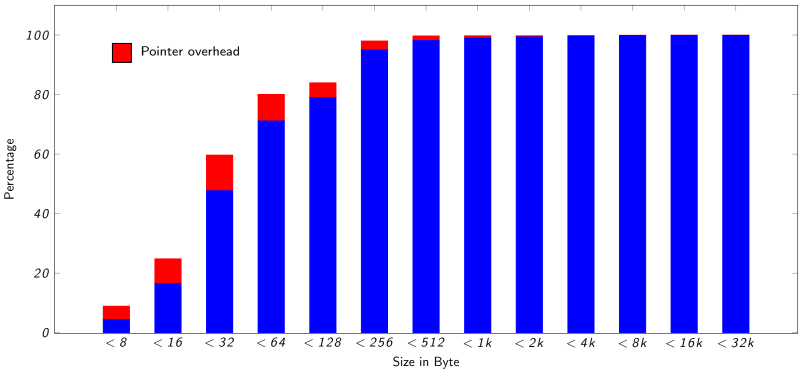
Looking at the size of allocations



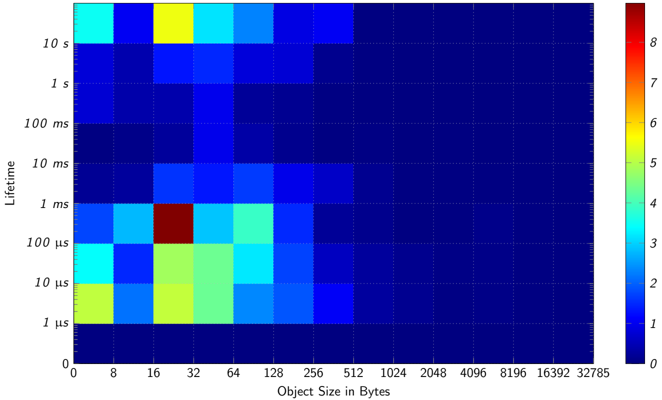
Object Sizes

Looking at the size of allocations

Number of certain allocation sizes compared to the total number of malloc calls during the main loop

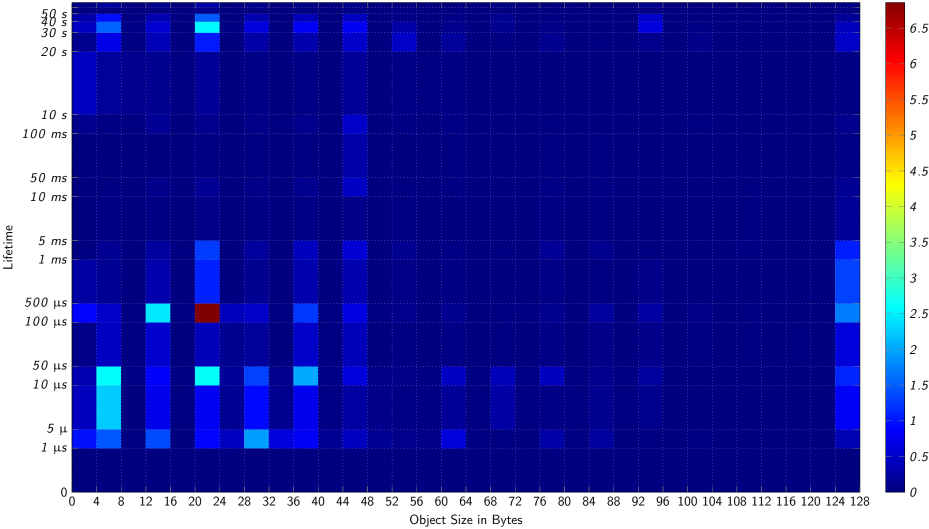


Object Size versus Object Lifetime



Roughly 8.9 % of the allocations happening during the main loop live between 100 μs and 1 *ms* and measure between 16 and 32 Bytes.

Object Size versus Object Lifetime



Sort Based Analysis

Sort the records by allocation size

Size in Bytes	Percentage
24	19.8%
8	11.2%
128	8.5%

Sort Based Analysis

Check most common stack trace chains

Size in Bytes	Percentage
24	19.8%
8	11.2%
128	8.5%

#	Stacktrace ID	Counts	Percentage	Bytes
1	350792 350790 350785 350975 350696 340151 340149 340150 340151 340149 340150 340151 ...	3,446,741	1.4%	24
2	139508 350689 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 ...	2,610,367	1.1%	8
3	340449 340445 340446 340437 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	24
4	350690 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	2,610,367	1.1%	96
5	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	8
6	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	24
7	350503 350691 350692 350671 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	8
8	350793 350794 350795 350796 350975 350696 340151 340149 340150 340151 340149 340150 ...	1,854,205	0.8%	24
9	351381 351620 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	1,699,944	0.7%	48
10	350792 350790 350785 350781 350696 340151 340149 340150 340151 340149 340150 340151 ...	1,638,489	0.6%	24

↑

Stack trace id mapping to source-line

Sort Based Analysis

Check most common stack trace chains

Mostly std::list nodes. Opportunity to improve memory layout and utilization

Size in Bytes	Percentage
24	19.8%
8	11.2%
128	8.5%

#	Stacktrace ID	Counts	Percentage	Bytes
1	350792 350790 350785 350975 350696 340151 340149 340150 340151 340149 340150 340151 ...	3,446,741	1.4%	24
2	139508 350689 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 ...	2,610,367	1.1%	8
3	340449 340445 340446 340437 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	24
4	350690 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	2,610,367	1.1%	96
5	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	8
6	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	24
7	350503 350691 350692 350671 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	8
8	350793 350794 350795 350796 350975 350696 340151 340149 340150 340151 340149 340150 ...	1,854,205	0.8%	24
9	351381 351620 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	1,699,944	0.7%	48
10	350792 350790 350785 350781 350696 340151 340149 340150 340151 340149 340150 340151 ...	1,638,489	0.6%	24

Stack trace id mapping to source-line

Sort Based Analysis

Check most common stack trace chains

Size in Bytes	Percentage
24	19.8%
8	11.2%
128	8.5%

Mostly std::list nodes. Opportunity to improve memory layout and utilization

Mostly 1x1 Eigen matrices and pointers to them in tracking. A design issue?

#	Stacktrace ID	Counts	Percentage	Bytes
1	350792 350790 350785 350975 350696 340151 340149 340150 340151 340149 340150 340151 ...	3,446,741	1.4%	24
2	139508 350689 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 ...	2,610,367	1.1%	8
3	340449 340445 340446 340437 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	24
4	350690 350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	2,610,367	1.1%	96
5	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	8
6	350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...	2,610,367	1.1%	24
7	350503 350691 350692 350671 340151 340149 340150 340151 340149 340150 340151 340147 ...	2,610,367	1.1%	8
8	350793 350794 350795 350796 350975 350696 340151 340149 340150 340151 340149 340150 ...	1,854,205	0.8%	24
9	351381 351620 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...	1,699,944	0.7%	48
10	350792 350790 350785 350781 350696 340151 340149 340150 340151 340149 340150 340151 ...	1,638,489	0.6%	24

Stack trace id mapping to source-line

Investigation lifetime plots with bokeh

Possible causes for patterns:

1. Curves: Accelerating loops
   - E.g. A lot of track candidates in the beginning and towards the end of the track more and more candidates are removed
2. Straight line: Linear loop that creates and deletes objects at similar timesteps
3. Ascending line: Objects (e.g. vector of elements with reserved size) created at the same time and then deleted element by element in a loop
4. Descending line: Objects (e.g. empty vector) extended step by step and then deleted in one go

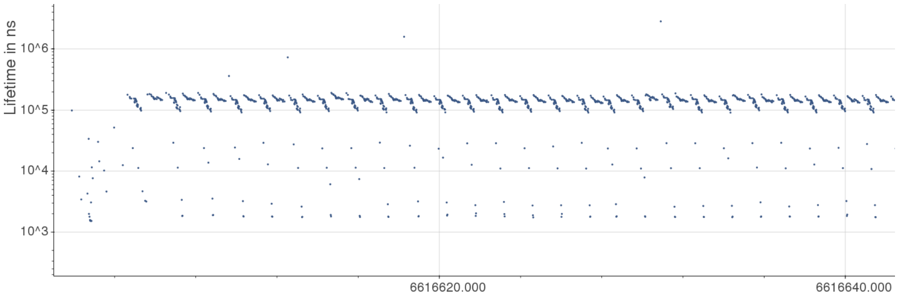
Figure: Pattern 1

Figure: Pattern 2

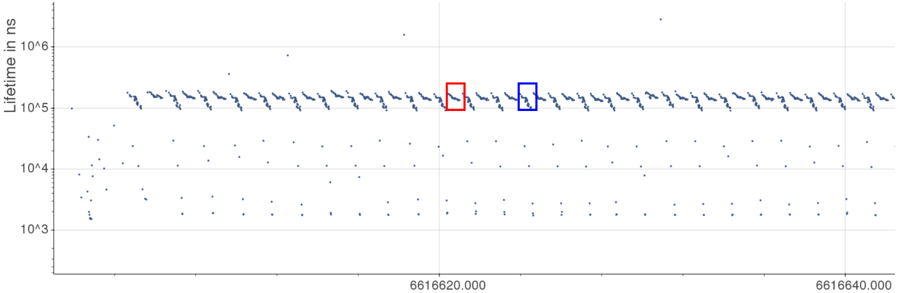
Figure: Pattern 3

Figure: Pattern 4

An Interesting Pattern in Lifetime Plots

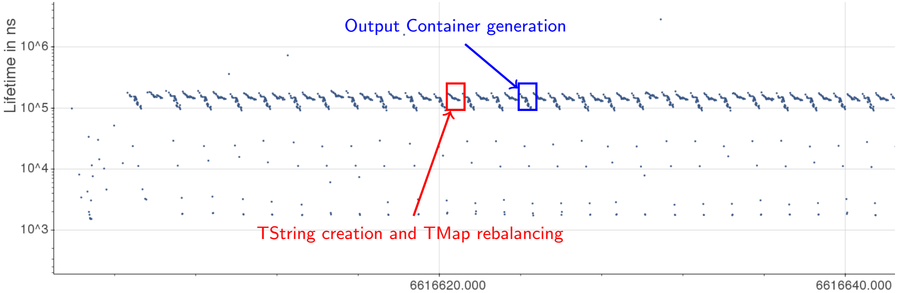


An Interesting Pattern in Lifetime Plots



Two repeating patterns

An Interesting Pattern in Lifetime Plots



Two repeating patterns

Origin of patterns

```
//In pseudo code
Root::TAccept selectionPassed;
for (itr in tracks) {
  if (some criteria){
    selectionPassed=m_trkFilter->accept(**itr,....);
  }
  if (selectionPassed)
  {
    //add track to output container
  }
}
```

← FOM identifed line. No apparent allocations

Origin of patterns

```
//In pseudo code
Root::TAccept selectionPassed;
for (itr in tracks) {
  if (some criteria){
    selectionPassed=m_trkFilter->accept(**itr,....);
  }
  if (selectionPassed)
  {
    //add track to output container
  }
}
```

← FOM identifed line. No apparent allocations

```
// accept() signature
Root::TAccept& TrkFilter::accept(....){/*...*/}
```

← Returns reference, no allocations. OK

Origin of patterns

```
//In pseudo code
Root::TAccept selectionPassed;
for (itr in tracks) {
  if (some criteria){
    selectionPassed=m_trkFilter->accept(**itr,....);
  }
  if (selectionPassed)
  {
    //add track to output container
  }
}
```

FOM identifed line. No apparent allocations

```
// accept() signature
Root::TAccept& TrkFilter::accept(....){/*...*/}
```

Returns reference, no allocations. OK

```
namespace Root{
  class TAccept{
    // Class methods
    //....
  private:
    TString m_memberString;
  };
}
```

Has a member TString!

Origin of patterns

```
//In pseudo code
Root::TAccept selectionPassed;
for (itr in tracks) {
  if (some criteria){
    selectionPassed=m_trkFilter->accept(**itr,....);
  }
  if (selectionPassed)
  {
    //add track to output container
  }
}
```

Cause a copy of TAccept i.e delete TString, remove TString from ROOT internal maps, construct a new TString, insert it into ROOT maps

```
// accept() signature
Root::TAccept& TrkFilter::accept(....){/*...*/}
```

Returns reference, no allocations. OK

```
namespace Root{
  class TAccept{
    // Class methods
    //....
  private:
    TString m_memberString;
  };
}
```

Has a member TString!

Origin of patterns

```
//In pseudo code
Root::TAccept selectionPassed;
for (itr in tracks) {
  if (some criteria){
    selectionPassed=m_trkFilter->accept(**itr,....);
  }
  if (selectionPassed)
  {
    //add track to output container
  }
}
```

← Cause a copy of TAccept i.e delete TString, remove TString from ROOT internal maps, construct a new TString, insert it into ROOT maps **For all tracks!**

```
// accept() signature
Root::TAccept& TrkFilter::accept(....){/*...*/}
```

← Returns reference, no allocations. OK

```
namespace Root{
  class TAccept{
    // Class methods
    //....
  private:
    TString m_memberString;
  };
}
```

← Has a member TString!

Example-2: Targeted Analysis

A simulation job of 50 events

- Generated roughly 746M records
- About 9GB compressed binary output file (with 50 deep stack traces)
- Targeting std::string(stream) related allocations
  - About 27.6M allocations related with strings (3.6% of total)
  - 385374 unique allocation traces
  - 530690 unique traces and sizes

Example-2: Targeted Analysis

A simulation job of 50 events

- Generated roughly 746M records
- About 9GB compressed binary output file (with 50 deep stack traces)
- Targeting std::string(stream) related allocations
  - About 27.6M allocations related with strings (3.6% of total)
  - 385374 unique allocation traces
  - 530690 unique traces and sizes
- Especially if you consider log file contains 412132 characters in 50403 words in 6042 lines.

String Hotspots

#	Stacktrace ID	Counts
1	1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...	4687515
2	1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...	4687515
3	1078 1079 1080 102889 102890 102884 10807 33149 102885	787187
4	1078 3559 3560 30120 102891 102890 102884 10807 33149 102885	787187
5	1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...	584465

String Hotspots

```
//pseudo code
for(ii:g4NavigationHistoryDepth){
  G4VPhysicalVolume* v1 = g4navigation->GetVolume(ii);
  // do some stuff...
  if (v1->GetName()==G4String(m_detectorName+"SomeSubDetName")) idep=ii;
  if (v1->GetName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
  // do more stuff ..
}
```

2 string constructions
+ forgotten code?

#	Stacktrace ID	Counts
1	1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...	4687515
2	1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...	4687515
3	1078 1079 1080 102889 102890 102884 10807 33149 102885	787187
4	1078 3559 3560 30120 102891 102890 102884 10807 33149 102885	787187
5	1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...	584465

String Hotspots

```
//pseudo code
for(ii:g4NavigationHistoryDepth){
  G4VPhysicalVolume* v1 = g4navigation->GetVolume(ii);
  // do some stuff...
  if (v1->GetName()==G4String(m_detectorName+"SomeSubDetName")) idep=ii;
  if (v1->GetName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
  // do more stuff ..
}
```

2 string constructions
+ forgotten code?

#	Stacktrace ID	Counts
1	1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...	4687515
2	1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...	4687515
3	1078 1079 1080 102889 102890 102884 10807 33149 102885	787187
4	1078 3559 3560 30120 102891 102890 102884 10807 33149 102885	787187
5	1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...	584465

Geometry
Construction

String Hotspots

```
//pseudo code
for(ii:g4NavigationHistoryDepth){
  G4VPhysicalVolume* v1 = g4navigation->GetVolume(ii);
  // do some stuff...
  if (v1->GetName()==G4String(m_detectorName+"SomeSubDetName")) idep=ii;
  if (v1->GetName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
  // do more stuff ..
}
```

2 string constructions
+ forgotten code?

#	Stacktrace ID	Counts
1	1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...	4687515
2	1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...	4687515
3	1078 1079 1080 102889 102890 102884 10807 33149 102885	787187
4	1078 3559 3560 30120 102891 102890 102884 10807 33149 102885	787187
5	1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...	584465

Geometry
Construction

String construction from const char*

Summary

- FOM-Tools provides detailed view of an applications allocation patterns
- Data can be analyzed in various ways
- Even simplest of analysis can reveal issues not apparent from source code
- Detailed analysis of the data can provide design hints and optimization opportunities

Thank you for your attention