Identifying Memory Allocation Patterns in HEP Software

Sami Kama\textsuperscript{1}, Nathalie Rauschmayr\textsuperscript{2}

\textsuperscript{1}Southern Methodist University
\textsuperscript{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

⇒ 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 negligible ($\ll 100 \, MB$)
- Typical output size, depending on the job and tool configuration is $O(10\, GB)$ for compressed, $O(100\, GB)$ 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

FOM Output → Import → hadoop → Export → Files → Bokeh

- Convert
- Filter/Select
- Create subsets and load into memory
- Query subset and visualize

Your favorite BigData analysis

ROOT

Your ROOT Analysis
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)

![Allocated Memory Diagram]

Only data after start of event loop are considered
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)

Only data after start of event loop are considered

Identifying memory allocation patterns in HEP software

S. Kama - CHEP 16
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)

![Graph showing allocated memory over job duration](image)

*Only data after start of event loop are considered*
Object Lifetimes

Looking at the lifetime of allocations

Object Lifetimes compared to the total number of allocations

Object lifetimes (including profiling overhead)

Percentage

< 1 µs < 10 µs < 100 µs < 1 ms < 10 ms < 100 ms < 1 s < 10 s < 100 s

More than 65% of allocations have lifetimes ≤ 10 ms ≪ event processing time
Object Lifetimes

Looking at the lifetime of allocations

Object Lifetimes compared to the total number of allocations

More than 65% of allocations have lifetimes $\leq 10$ ms $\ll$ event processing time
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

Size in Byte

Percentage

0 20 40 60 80 100

< 8 < 16 < 32 < 64 < 128 < 256 < 512 < 1k < 2k < 4k < 8k < 16k < 32k
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

- Pointer overhead

Size in Byte

- < 8
- < 16
- < 32
- < 64
- < 128
- < 256
- < 512
- < 1k
- < 2k
- < 4k
- < 8k
- < 16k
- < 32k

Percentage
Object Size versus Object Lifetime

Roughly 8.9% of the allocations happening during the main loop live between 100 $\mu$s and 1 ms and measure between 16 and 32 Bytes.
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.
Sort Based Analysis

Sort the records by allocation size

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>19.8%</td>
</tr>
<tr>
<td>8</td>
<td>11.2%</td>
</tr>
<tr>
<td>128</td>
<td>8.5%</td>
</tr>
</tbody>
</table>

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

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

Check most common stack trace chains

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>19.8%</td>
</tr>
<tr>
<td>8</td>
<td>11.2%</td>
</tr>
<tr>
<td>128</td>
<td>8.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Stacktrace ID</th>
<th>Counts</th>
<th>Percentage</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>350792 350790 350785 350975 350696 340149 340150 340151 340149 340150 340151 ...</td>
<td>3,446,741</td>
<td>1.4%</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>139508 350689 350671 340149 340149 340105 340151 340149 340150 340151 340147 340148 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>340449 340445 340446 340437 340149 340150 340151 340149 340150 340151 340149 340147 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>350690 350671 340149 340150 340151 340149 340147 340149 340151 340147 340148 327958 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>350671 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 327736 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>350503 350691 350692 350671 340149 340150 340151 340149 340150 340151 340147 340148 ...</td>
<td>2,610,367</td>
<td>1.1%</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>350793 350794 350795 350796 350975 350696 340151 340149 340150 340151 340149 340150 ...</td>
<td>1,854,205</td>
<td>0.8%</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>351381 351620 340151 340149 340150 340151 340149 340150 340151 340147 340148 327958 ...</td>
<td>1,699,944</td>
<td>0.7%</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>350792 350790 350785 350696 340151 340149 340150 340151 340151 340149 340148 327958 ...</td>
<td>1,638,489</td>
<td>0.6%</td>
<td>24</td>
</tr>
</tbody>
</table>

Stack trace id mapping to source-line
Sort Based Analysis

Check most common stack trace chains

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>19.8%</td>
</tr>
<tr>
<td>8</td>
<td>11.2%</td>
</tr>
<tr>
<td>128</td>
<td>8.5%</td>
</tr>
</tbody>
</table>

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

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

Stack trace id mapping to source-line
Sort Based Analysis

Check most common stack trace chains

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>19.8%</td>
</tr>
<tr>
<td>8</td>
<td>11.2%</td>
</tr>
<tr>
<td>128</td>
<td>8.5%</td>
</tr>
</tbody>
</table>

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

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

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

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

![Graph showing lifetime plots with a repeating pattern.](image)
An Interesting Pattern in Lifetime Plots

Two repeating patterns
An Interesting Pattern in Lifetime Plots

Two repeating patterns

Output Container generation

TString creation and TMap rebalancing

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
    }
}
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
    }
}

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

FOM identified line. No apparent allocations.

Returns reference, no allocations. OK
Origin of patterns

```cpp
// 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
    }
}

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

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

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

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

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

Returns reference, no allocations. OK

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
    }
}

// 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

<table>
<thead>
<tr>
<th>#</th>
<th>Stacktrace ID</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>2</td>
<td>1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>3</td>
<td>1078 1079 1080 102889 102890 102884 102885 33149 102885 787187</td>
<td>787187</td>
</tr>
<tr>
<td>4</td>
<td>1078 3559 3560 30120 102891 102890 102884 10807 33149 102885 787187</td>
<td>787187</td>
</tr>
<tr>
<td>5</td>
<td>1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...</td>
<td>584465</td>
</tr>
</tbody>
</table>
String Hotspots

2 string constructions + forgotten code?

```cpp
//pseudo code
for(ii:gNavigationHistoryDepth){
    G4PhysicalVolume* vl = gNavigation->GetVolume(ii);
    // do some stuff...
    if (vl->GetName()==G4String(m_detectorName+"SomeSubDetName")) idep=ii;
    if (vl->GetName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
    // do more stuff ..
}
```

<table>
<thead>
<tr>
<th>#</th>
<th>Stacktrace ID</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>2</td>
<td>1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>3</td>
<td>1078 1079 1080 102889 102890 102884 10807 33149 102885</td>
<td>787187</td>
</tr>
<tr>
<td>4</td>
<td>1078 3559 3560 30120 102891 102884 10807 33149 102885</td>
<td>787187</td>
</tr>
<tr>
<td>5</td>
<td>1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...</td>
<td>584465</td>
</tr>
</tbody>
</table>
String Hotspots

2 string constructions + forgotten code?

```cpp
// pseudo code
for(ii;ii>NavigationHistoryDepth){
  G4PhysicalVolume* vl = g4navigation->GetVolume(ii);
  // do some stuff...
  if (vl->GetName()==G4String(m_detectorName+"SomeSubDetName")) idep=ii;
  if (vl->GetName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
  // do more stuff ..
}
```

<table>
<thead>
<tr>
<th>#</th>
<th>Stacktrace ID</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1078 1493 1494 3487 3488 123384 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>2</td>
<td>1078 1493 1494 3487 3488 123385 123297 123298 123239 123218 122997 ...</td>
<td>4687515</td>
</tr>
<tr>
<td>3</td>
<td>1078 1079 1080 102889 102890 102884 10807 33149 102885</td>
<td>787187</td>
</tr>
<tr>
<td>4</td>
<td>1078 3559 3560 30120 102891 102890 102884 10807 33149 102885</td>
<td>787187</td>
</tr>
<tr>
<td>5</td>
<td>1078 1079 1080 124549 124547 123297 123298 123239 123218 122997 ...</td>
<td>584465</td>
</tr>
</tbody>
</table>
String Hotspots

2 string constructions + forgotten code?

```
//pseudo code
for(ii:g4NavigationHistoryDepth){
    G4PhysicalVolume* vl = g4navigation->GetVolume(ii);
    // do some stuff...
    if(vl->GetFileName()==G4String(m_detectorName+"SomeSubDetName")) testbeam=true;
    if(vl->GetFileName()==G4String(m_detectorName+"AnotherSubDetName")) testbeam=true;
// do more stuff ..
}
```

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

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