Identifying Memory Allocation Patterns in HEP Software

To cite this article: S Kama and N Rauschmayr 2017 J. Phys.: Conf. Ser. 898 072031

View the article online for updates and enhancements.

Related content
- Achieving production-level use of HEP software at the Argonne Leadership Computing Facility
  T D Uram, J T Childers, T J LeCompte et al.
- Developing and Optimizing Applications in Hadoop
  P Kothuri, D Garcia and J Hermans
- Hadoop and friends - first experience at CERN with a new platform for high throughput analysis steps
  D Duellmann, K Surdy, L Menichetti et al.
Identifying Memory Allocation Patterns in HEP Software

S Kama\(^1\) and N Rauschmayr\(^2\)

\(^1\) Department. of Physics, Southern Methodist University, Dallas, TX, USA
\(^2\) CERN, Geneva, Switzerland

E-mail: sami.kama.cern@gmail.com

Abstract. HEP applications perform an excessive amount of allocations/deallocations within short time intervals which results in memory churn, poor locality and performance degradation. These issues are already known for a decade, but due to the complexity of software frameworks and billions of allocations for a single job, up until recently no efficient mechanism has been available to correlate these issues with source code lines. However, with the advent of the Big Data era, many tools and platforms are now available to do large scale memory profiling. This paper presents, a prototype program developed to track and identify each single (de-)allocation. The CERN IT Hadoop cluster is used to compute memory key metrics, like locality, variation, lifetime and density of allocations. The prototype further provides a web based visualization back-end that allows the user to explore the results generated on the Hadoop cluster. Plotting these metrics for every single allocation over time gives a new insight into application’s memory handling. For instance, it shows which algorithms cause which kind of memory allocation patterns, which function flow causes how many short-lived objects, what are the most commonly allocated sizes etc. The paper will give an insight into the prototype and will show profiling examples for the LHC reconstruction, digitization and simulation jobs.

1. Introduction

The cost gap between memory and CPU has constantly risen in the past decade. The cost of ownership of memory per GB dropped significantly over the years, however amount of data that needs to be processed have increased exponentially, requiring ever more memory resources and leading to higher power envelopes for servers and data centers. Many-core CPU’s introduced in the last few years encouraged proliferation of multi-threaded applications and with the arrival of big data wave, the stress on the CPU-Memory bandwidth increased proportionally. As a consequence, efficient memory handling became more important than ever before. This subject is also often discussed within the C++ committee [1].

The Large Hadron Collider (LHC) [2] at CERN near Geneva, Switzerland hosts four detectors that collect data from high energy proton-proton collisions. These detectors provide around 100 TB of data per second which is filtered and recorded for offline processing at roughly 4 GB/s rate. Total recorded data reaches about 30 PB per year. Because of limited computing budgets and complexity of the data processing, efficient resource utilization, especially memory consumption, is crucial for the analysis of the data.

There are many tools to study CPU performance but only a few can study the memory related performance issues. Valgrind [3] suite contains a few memory related tools but they...
are limited in data they expose. In order to study memory allocation patterns we developed FOM-Tools. Using FOM-Tools we investigated large HEP frameworks, where many developers with different experience levels contribute, to assess the memory performance. The paper shows that all profiled HEP applications suffer significantly from memory churn, meaning that 80% of allocations are smaller than 100 Bytes and about 65% live less than 1 ms.

2. FOM-Tools
FOM-Tools has two modes of operation. It can detect unused large allocations, which are valid allocations and missed by memory checkers, or it can record (de-)allocation calls in a process to identify allocation patterns of the process.

2.1. Unused memory detection
In unused memory detection mode, the tool starts the process under investigation in a memory constrained control group (cgroup) [4, 5]. A memory controlled cgroup limits the memory available to processes and forces them to swap if processes try to allocate beyond limit and a swap file or partition is available. Since accessing swapped pages are several orders of magnitude slower than accessing memory, it will lead to an increase in execution time of process depending on the amount of swapped pages accessed by the process. Thus if execution time of a memory constrained process is not changing significantly, it is an indication that process has unused or seldomly accessed pages in memory.

In the FOM-Tools execution, the user defines the amount of memory allocated to cgroup. This limit is usually found by executing the process with different cgroup memory thresholds until a memory sweet-spot is found such that execution time of the process slows down minimally, preferably in the order of a few percent. During the constrained execution, FOM-Tools collects memory allocation information by interposing malloc family of system calls and records the time of allocation, address returned by the original function, size of allocation and stack trace. At the same time, it periodically freezes the process using Freeze feature of cgroups and investigates page map of the process to identify the pages that are in swap and that are in memory. It can identify unused allocations and point to the source line that made it by analyzing these records and mapping them to the swapped pages, especially to the pages that stay in the swap throughout the execution of the process.

This mode has been successfully used to identify around O(200MB) allocations that were used as de-serialization buffers and kept until the end of the job in several HEP applications.

2.2. Allocation pattern identification
In the allocation pattern identification mode FOM-Tools do not use cgroups and collects all memory allocation operations by interposing malloc, calloc, realloc and free calls during the execution by interposing these functions. Whenever any of these functions are called, it records the time, the address returned or passed to the original function, function type, size requested and stack trace. It also records the time before serialization of record to correct for the overhead.

Depending on the amount of allocations and deallocations in the process and stack trace depth, run-time overhead is about of 5 to 6 times of native execution run-time. Memory overhead is negligible (≪ 100 MB) and typical output is O(10GB) for compressed output and O(100GB) for uncompressed output.

Then the FOM-Tools records are post-processed to calculate

- Density: Amount of allocations and deallocations per unit time.
- Variation: Size difference between consecutive allocations.
- Locality: Proximity of returned addresses in consecutive allocations.
- Lifetime: Time difference between an allocation and respective free call.
These metrics show allocation behavior of the code and hint possible optimization opportunities. For example, density is a measure of allocations and deallocations in a given unit time. If the density is high, the application is doing frequent allocations and deallocations and an arena approach might be beneficial. On the other hand, variation is a measure of how allocation sizes differ. If variation is zero for some time, it may mean that the application is allocating same sized objects over and over again. If possible, a bulk allocation of such objects or a memory arena may improve performance. Moreover repeating patterns in variation may point to a design issue. Locality is a metric for the allocator. A good allocator should preferably have low locality values. The locality metric can be used in optimization of the allocator library. Lifetime is a measure of the duration each allocation is active. Temporary objects in processing usually have smaller lifetimes. Lifetime metric is usually the most indicative of the improvement opportunities. Examples in this paper focus on lifetime based analysis.

2.3. Analysis

FOM-Tools metrics can be calculated and analyzed in various ways. Figure 1 shows possible analysis and post-processing paths that can be taken. Examples given in this paper are analyzed using the methods demonstrated in the figure.

First method involves importing data to a Hadoop [6] cluster and then doing filtering and selection with Zeppelin [7] notebooks. Once the selection and filtering is complete, the data is exported to a set of files which are then read into Bokeh [8] to create visualizations through a web browser. Snapshot of an example session can be seen in figure 2. The second method involves converting FOM-Data to a ROOT [9] file. Since many high energy physicist and programmers are familiar with ROOT, this allows an easy implementation of their own analysis. Finally the FOM-Tools data can be exported to a CSV file to be imported to Big Data analysis tools. Depending on the available resources and preferred analysis technique, an appropriate method can be chosen.

3. Example Analyses

In this section analyses of FOM-Tools data from a reconstruction and a simulation job are presented. Typically reconstruction and simulation tasks comprise more than 75% of the total CPU workload on the LHC computing grid.
Figure 2: A snapshot of interactive display produced from FOM-Tools data. Tool-tips give additional details on each data point.
3.1. General Exploration

An event reconstruction job of 50 simulated events from one of the Big LHC experiments has been analyzed. For this profiling, FOM-Tools created a 90 GB uncompressed file containing about 870 M records with 20 deep stack traces. In the analysis, only the data after the beginning of event loop is considered. About 65% of the allocations are found to have a lifetime less than 10 ms which is much less than the average event processing time of about a minute. Moreover approximately 70% of the allocations were below 64 bytes, leading to about 22% memory overhead due to pointers. Plots of lifetime and size distributions can be seen in figure 3 left and right, respectively.

Investigation of lifetime versus object size plot, figure 4, exposes a hot spot that is caused by allocations between 16 bytes and 32 bytes in size and live shorter than 1 ms but longer than 100 µs. The plot also shows that the bulk of the allocations are small in size and only a few percent of the allocations are actually kept longer than a few seconds. The gap in the lifetime axis reveals that the most of the allocations are due to temporary objects that are discarded immediately and a very small percentage of these allocations are actually towards final data that are saved to the disk.

Examining allocation data with respect to allocation sizes we found that the top three allocation sizes were 24, 8, 128 bytes with 19.8%, 11.2%, and 8.5% of the total record count, respectively. Upon investigation, bulk of 24 byte allocations were due to `std::list` nodes of pointers. Large portion of 8 byte allocations were done by 1x1 Eigen [10] matrices and pointers to them which are allocated on heap. Both of these are related to design choices and can be improved.

Visual investigation of lifetime distributions also revealed interesting patterns. One of the many interesting patterns, a sequence of repeating consecutive allocations that has lifetimes less than 20 µs, is shown in figure 5. Stack traces of the allocations reveal that the points in the blue box are generated due to the creation of output containers for a certain algorithm during the event loop. On the other hand, the points in the red box are due to the creation of ROOT’s `TString` objects and respective re-balancing of internal `TMap` structure.

Source lines that are identified by FOM-Tools which are responsible of these patterns are listed, in pseudo-code, in listing 1. FOM-Tools points to line 6 of the left listing as the cause of rapid allocations and deallocations. Implementation of the `accept` method does not allocate memory and returns a reference to a member `TAccept` object. Declaration of `TAccept` class
Figure 4: Allocation percentages in object size versus lifetime. Hot spot in bin with $100 \mu s < \text{lifetime} < 1\text{ms}$ and $16\text{B} < \text{size} < 32\text{B}$ is clearly visible. Plot shows that most of the allocations are short lived and a few bytes in size.

Figure 5: An interesting pattern in lifetime plot of allocations. The pattern in the red box is due to $TString$ creation and re-balancing of ROOT’s $TMap$ data structure. The pattern in the blue box is due to creation of out containers of the job.
contains a private member \textit{ TString} object. When the \textit{ accept} method returns the reference, a copy of the returned object replaces the \textit{ selectionPassed} variable created on stack on line 3. During the destruction of the old object, \textit{TString} is removed from the global object map of ROOT and during the copy, a new object is inserted. All these copy/delete operations are done in a very tight loop forcing frequent re-balancing of the map and very short lifetime allocations.

3.2. Targeted Analysis

It is possible to use FOM-Tools data to do a targeted analysis where allocations due to a specific type of class, groups of classes, certain stack traces or code locations can be investigated.

In this example 50 events processed with a full detector simulation based on Geant4 [11] are profiled, generating a 9 GB compressed format file containing about 746 M records with up to 50 deep stack traces at every record.

Strings are one of the constructs in programs that are typically used lavishly since they make sense for humans and provide very little real use for computers. Thus for this example, allocations and deallocations due to \textit{std::string} and \textit{std::stringstream} operations are analyzed. About 3.6\% of total, in other words 27.6 M records were due to string related allocations. These allocations were scattered over more than 385 k unique stack traces. Considering the fact that the log file output contains around 50 k words in roughly 6 k lines, these numbers are an indication for a potential for improvement.

Top 2 most frequently encountered sources of string allocations are given in pseudo-code in listing 2. Lines 6 and 7 are called about 4.7 M times, creating extremely short lived temporary string objects. These lines are probably remains from early development days and could be
replaced by a more efficient version or even removed completely. Other frequent allocation sources are found to be due to naming of detector elements during detector geometry construction and temporary string construction from a `const char` pointer argument being passed to a function with a non-const string argument.

4. Conclusions
Understanding memory utilization of a process is important and memory leaks are not the only problems that can arise from poor memory utilization practices. Profiled HEP applications has shown large amounts of small allocations that can lead to memory churn, which can be replaced, as well as allocations that can be removed completely. The FOM-Tools provides a deep insight into memory allocation patterns of applications. It can identify easy to miss performance problems, provide hints for design and allocator choices. Examples in this paper are a small demonstration of its capabilities. Further study of the data might reveal further performance improvement opportunities. The FOM-Tools can be downloaded from the github repository [12]. There are ongoing studies towards producing a tool for the Valgrind suite with the FOM-Tools capabilities.

References
[2] Evans L and Bryant P 2008 JINST 3 S08001