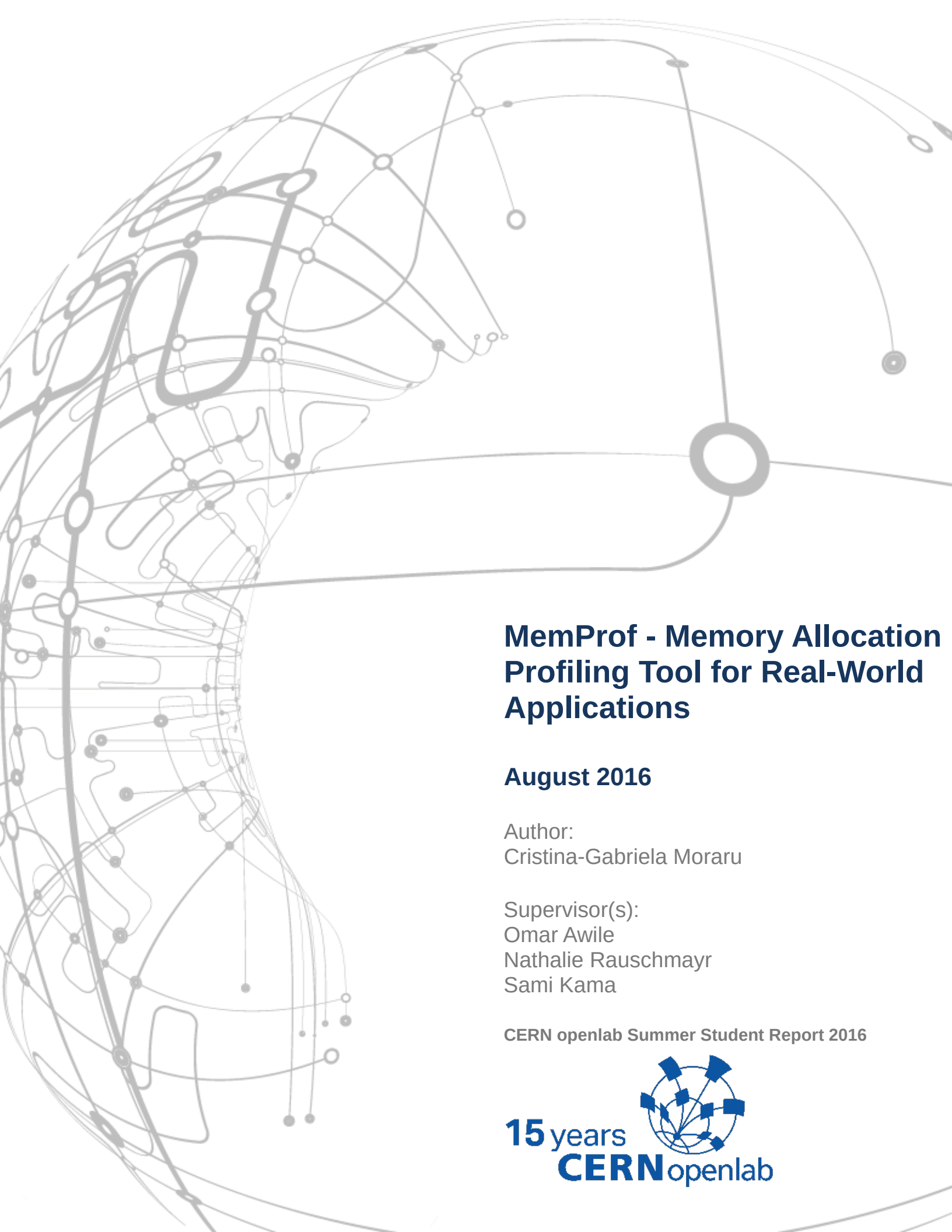

MemProf - Memory Allocation Profiling Tool for Real-World Applications

August 2016

Author:
Cristina-Gabriela Moraru

Supervisor(s):
Omar Awile
Nathalie Rauschmayr
Sami Kama

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

This project aims to build a memory allocation profiling on top of the popular dynamic analysis framework Valgrind, able to detect heap memory waste or misusage. The project is divided in several components:

1. Track allocations / deallocations of the client program – create a Valgrind plug-in tool able to detect when the target program performs a heap memory operation (allocation / deallocation)

2. Collect and store relevant information about the allocations / deallocations – for each operation keep the call parameters and other meta-information about its context.

3. Implement compression on the output analysis data – the tool should provide an option to output compressed analysis data instead of plain text.

The result of this project is a plug-in tool for Valgrind which creates memory usage statistics for any real-world application. This project aims to be part of Valgrind official repository.

Abstract

The Large Hadron Collider (LHC) experiments produce a vast amount of data and its throughput is even increasing in time. Scalability is the main keyword and adding more hardware often solves the problem but is never the most cost efficient solution. Technology permitted us to build increasingly stronger CPUs and there are several tools that aid in understanding their utilization and suggest possible optimizations. The memory represents the main bottleneck since its bandwidth is limited and there are only few tools focusing on profiling memory efficiency. MemProf is a memory allocation profiling tool built on top of Valgrind, a very popular open source dynamic analysis framework, thereby taking benefit both from already existing components and high maintenance from its large number of contributors. Moreover, Valgrind's architecture permits the tool to access more fine-grained features such as support for multi-threaded programs and memory access tracking. The tool has been run over several standard Linux programs such as unzip, telnet, netstat, evince and is currently optimized for analysing larger applications. This paper presents the development process of this project, the current status and potential future extensions.

Table of Contents

Contents

1 Introduction

The Large Hadron Collider (LHC) experiments produce on average 1Pb raw data per second and the majority of it describes background events collected by the sensors but that are irrelevant for research. The Data Acquisition Systems of LHC experiments perform intensive hardware and software filtering in real time so that only potentially interesting data is kept and sent further to offline analysis. After the last processing phase the resulting data is ~ 25 PB/year, which also represents the amount that is stored [1]. This data might lead to the discovery of new particles or a better understanding of the Standard Model so the two big requirements for the software behind online filtering and offline processing are correctness and speed. To achieve these targets, the current implementation sums up a few millions lines of code and with such a code base there is always room for optimization. Also, during the online analysis, processing introduces a latency of about 3 μs which becomes significant when compared to the frequency of the acquired data of a few nanoseconds [1].

One strategy to increase data-processing throughput is to add more hardware resources, however, this is often not the most cost efficient solution. Since the evolution to multi-core, CPU processing power has increased considerably technology now permits us to increase parallelism. Additionally, a number of benchmarking tools exist, which aid in understanding application CPU utilization and give hints of possible optimizations. The greatest bottleneck we face is the memory. Unlike the CPU, there are few tools focusing on memory analysis and most of them target finding errors such as memory leaks or illegal memory accesses. In this project, however, we are mostly interested in inefficient use of memory.

MemProf is a memory allocation profiling tool meant to detect possible memory waste and is built on top of Valgrind, a popular open source Dynamic Binary Analysis (DBI) framework. Relying on Valgrind as base technology permitted us making use of already implemented components and of continuous maintenance provided by its large number of contributors. A previous attempt has been made in this direction: FOM-Tools was developed at CERN in order to find unused memory allocations, lifetimes and patterns but it is not currently possible to track correctly memory allocations within multiple threads or analyze memory access patterns. MemProf solves these shortcomings and provides support for multi-threaded applications because Valgrind ensures correct thread serialization. Moreover, it can be extended to more fine-grained analysis such as memory access tracking.

This paper presents MemProf, a new memory allocation profiling tool built on top of Valgrind. Chapter 2 describes the state of the art, namely a previous similar approach FOM-Tools and the base technology of this project, Valgrind. Chapter 3 we discuss MemProf's design and implementation details. Chapter 4 presents some initial results obtained with MemProf Finally, chapter 6 summarizes the work done within this project and give some ideas about possible future development.

2 State of the Art

2.1 FOM-Tools

This chapter will describe the already existing technology FOM-Tools in order to provide a better understanding of how MemProf is different and what its advantages are. .

FOM-Tools (Find Obsolete Memory) [2] aims to detect unused memory by finding pages from the process address space that move to swap and remain there. The objects contained in these pages are identified as being obsolete and could be released without affecting the execution.

The workflow of FOM-Tools is summarized in Figure 1. A series of hooks are implemented for each memory allocation / deallocation operation which analyse the placement of the current object within the address space. The Linux Kernel feature Control Groups (Cgroups) [3] are used for a fine-grained control over the memory management which allows restricting jobs to run with much less memory footprint [4]. After each memory operation, the process is "frozen" and the virtual addresses of the objects are searched in the process address space in order to detect if they belong to a page in RAM or SWAP.

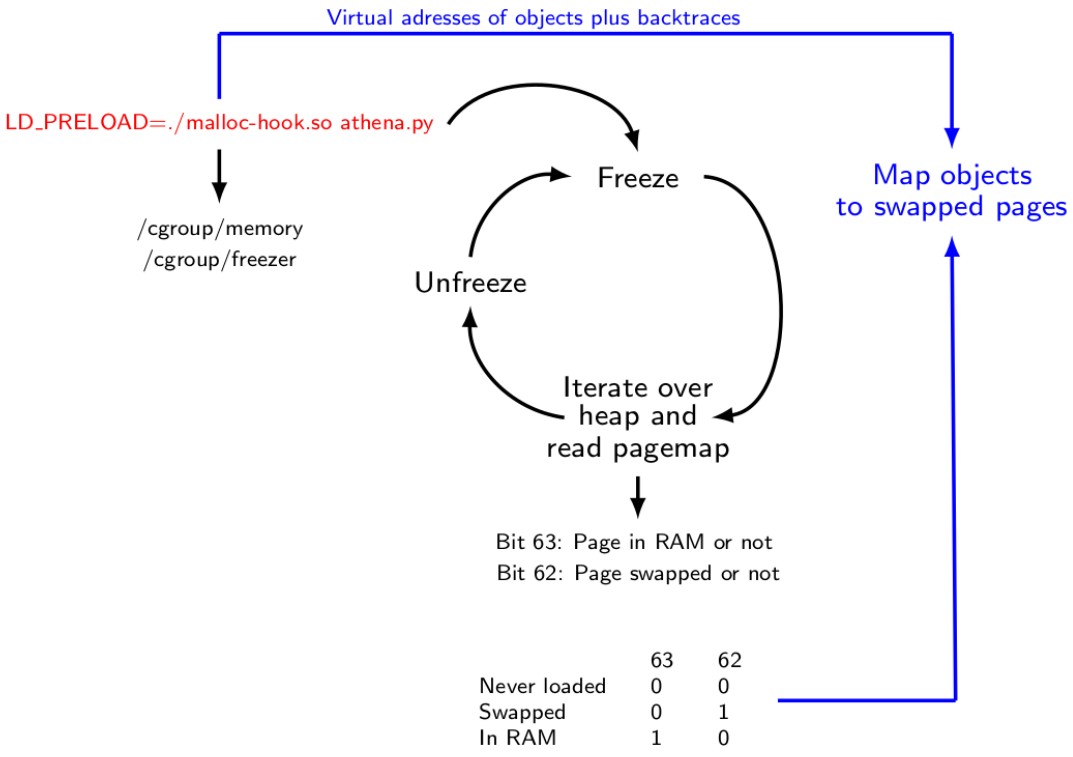

Figure 1. FOM-Tools analysis workflow.

The pagemap [5] is a set of interfaces in the kernel that expose to userspace programs page tables or other related information in the virtual file system /proc. In other words, each process has its own pagemap represented in an associated file:

/proc/<PID>/pagemap

The content of this file maps each virtual page to the corresponding physical address. For each virtual page there is an associate 64-bit value structured as follows:

Table 1. Structure of a 64-bit entry in /proc/<PID>/pagemap

Bits	Role
0-54	page frame number (PFN) if present
0-4	swap type if swapped
5-54	swap offset if swapped
55	pte is soft-dirty
56	page exclusively mapped
57-60	zero
61	page is file-page or shared-anon
62	page swapped
63	page present

As can be seen both in in Table 1 and Fig. 1 the bits of interest are 62 and 63 which identify the location of the virtual page. Shortly, after each memory operation, FOM-Tools iterates over all heap addresses and reads the pagemap in order to detect the location of the objects (RAM vs. SWAP).

Currently, FOM-Tools is not able to correctly track memory allocations in multi-threaded applications and is restrictive regarding extensions to more fine-grained analysis of memory allocation and accesses. These shortcomings are overcome in MemProf as we will elaborate in the following chapters.

2.2 Valgrind

This chapter will describe the architecture and workflow of Valgrind, the base technology of this project.

Valgrind [6] appeared as a novelty in Dynamic Binary Analysis (DBA) field as being a heavyweight tool by taking full advantage of Dynamic Binary Instrumentation (DBI) method unlike similar frameworks which focus on performance. It implements a unique technique of 'value shadowing' which requires doubling the memory size by keeping copies of every register and memory value and updating them at each change of state. Valgrind analysis add a slowdown of 10-100% ut its capabilities cannot be reproduced with other DBI frameworks such as Pin, DynamoRIO.

2.2.1 Architecture

Valgrind's architecture is modular which permits reusability of certain components in the development process as well as easy extensibility. The main component is the core (Coregrind) which performs the common tasks to all valgrind tools such as client program loading, instruction translation from ELF format to Valgrind intermediary representation (Valgrind IR), error handling, logging etc. A number of tools, which connect as plug-ins to the core and instrument the client accordingly, provide the required functionality for the different analysis types.

We enumerate 3 tools which Valgrind offers out of the box:

- memcheck is a memory error detector which shadows the client program's operations in order to encounter memory leaks, attempts of accessing undefined or unaccessible memory, mismatched allocations / deallocations, invalid address / size parameters.
- cachegrind analyses the interaction with the machine's cache hierarchy and, optionally, branch branch prediction
- callgrind builds the call graph profiles the runtime behaviour by collecting data about the number of instructions executed, relationship caller / callee, optional branch prediction

This architecture permitted the creation of a new plug-in tool for memory profiling by taking advantage of the facilities provided by the already implemented core.

Valgrind instrumentation is started using the command:

valgrind --tool=<tool-name> <client-name> [<client-parameters>]

The client program can be any program whose image is an ELF executable.

2.2.2 Execution flow

The previous command starts the Valgrind core which performs the general setup and then executes the tool within the same process. During the entire Valgrind instrumentation no child process is forked for the client program being instrumented. The client program is not loaded by the OS loader as a regular process but, instead, is mapped at a fixed address into Valgrind's address space and its execution is emulated.

The execution flow does not follow the standard path of a process since Valgrind does not link to the libc standard library. In normal process execution the entry point is the predefined function _start, which in turn calls the user defined main() function. Valgrind reimplements _start such that it first jumps to an architecture specific _start_<arch> function to perform additional setup before calling main(). Also, standard library functions such as open, close, malloc, or memcpy use in their implementation predefined addresses which are not cannot be used for two processes sharing the same address space. Consequently, they exist in two versions both for Valgrind and for client code.

3 Technical Implementation

This chapter will describe the key points of MemProf's design and implementation, will describe its connection with the Valgrind core, the interception and handling of memory operations, and the collected data and the output format.

3.1 Creating a new tool

As mentioned above, MemProf is a memory allocation profiler built as a Valgrind plug-in tool taking advantage of the facilities provided by the core. Figure 2 presents conceptually where MemProf is placed within Valgrind's architecture:

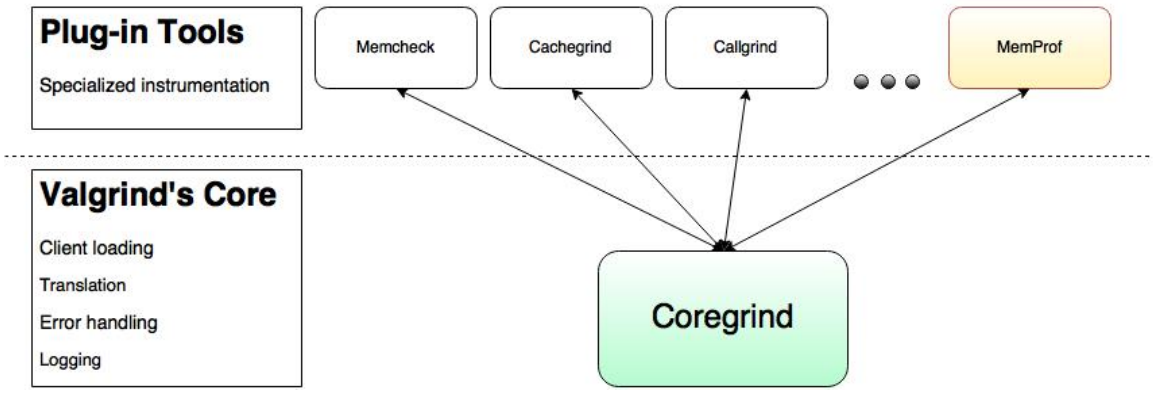

Figure 2. Integrating MemProf into Valgrind architecture

More concrete, each tool's implementation is placed in a directory with the same name in the root of Valgrind's source tree. The main implementation including the entry points must be placed in a specific file. In the case of MemProf this is mp_main.c. Of course auxiliary compilation files can be added for better code modularization. Four key callbacks are implemented in our tool:

- pre_clo_init - entry point run before the processing of the command line options
- post_clo_init - initialization done after processing the command line options
- instrument - instrumentation of the client code
- fini - callback for the end of instrumentation

Adding a new empty tool and recompiling enables Valgrind to recognize:

valgrind --tool=memprof <client-name> [<client-parameters>]

as a valid command but no instrumentation will be done until the instrument function has been implemented.

3.2 Tracking allocations / deallocations

Valgrind's core offers support for replacing memory operations such as allocations and deallocations with custom defined callbacks in order to permit additional processing for these events.

MemProf implements replacement wrappers for the following functions:

- malloc
- __builtin_new - instantiating an object in C++ with new operator
- __buitin_vec_new - instantiating a vector of objects in C++ with new operator
- memalign - basic allocation with extra parameter for custom alignment
- calloc
- free
- __builtin_delete- free an object created with the new operator in C++
- __builtin_vec_delete - free a vector of objects created with the new operator in C++

MemProf implements generic allocation and deallocation wrappers (handle_alloc / handle_free) which record the desired analysis information and call the original function. The function realloc is a special case because it creates two records: one for the allocation and one for the internal deallocation at the original address, therefore it has its own specialized wrapper handle_realloc.

3.3 Meta-Information

As mentioned previously, within the allocation / deallocation wrappers, additional information is collected about these operations. This meta-information is gathered in a structure _MP_Record which is presented in Figure 3 along with its composing fields:

```
struct _MP_Record {
   Addr                addr;
   SizeT               szB;
   ThreadId            tid;
   UShort              allockind;
   UShort              nips;
   ULong               begin_t, end_t, stack_t;
   UInt                *ips;
};
```

Figure 3. MemProf structure describing a record

The first field refers the memory address of the (de)allocated block. In case of an allocation, it's the address is returned by malloc while at deallocation, it is the parameter passed to the free call. The second field represents the size in bytes occupied by the (de)allocated block. For allocations it is received by the wrapper as parameter but for deallocations an internal hashmap is kept in order

to determine the size of a block placed at a given address. The Valgrind core performs thread serialization and identifies the thread ID which is passed as parameter to the wrappers.

Next the type of the memory operation is given by the index stored in the field allockind. The mapping is presented in Table 2:

Table 2. Indexes of memory operations

Operation	Index
malloc	0
calloc	1
realloc	2
new	3
new vector	4
free	5
delete	6
delete vector	7

The fields begin_t, end_t and stack_t are timestamps of the beginning of memory operation, end of memory operation and end of generating the context stacktrace. Implementing a function for calculating the timestamp in nanoseconds has been one of the requirements of this project.

In order to determine the location of a certain memory operation it is important to keep its context callstack. However, since some instruction pointers (IPs) from different stacktraces may coincide, it is space consuming to keep them for each record they appear in. Therefore, an additional hashmap is introduced to associate the instruction pointers (IPs) with an index. When an instruction pointer is encountered for the first time, it is assigned an unique index and the pair <IP, INDEX> is inserted into the hashmap. The _MP_Record's ips field represents the series of indexes corresponding to the IPs from the callstack and not the actual IPs. The fields nips gives the exact length of this array.

As can be seen in Figure 3, Valgrind has a number of internally defined data types. This was done both for better control of the variable sizes as well as for ease of readability. Valgrind types map standard types as follows:

Table 3. Mapping between Valgrind internal defined types and standard types

Valgrind type	Standard type	Size (bits)
Addr	unsigned long	32 / 64 (arch dependent)
SizeT	unsigned long	32 / 64 (arch dependent)
ThreadId	unsigned int	32
UShort	unsigned short	16
UInt	unsigned int	32
ULong	unsigned long long int	64

From table 3 we see that two size fields exist and the choice is architecture dependent. Considering that these fields refer to the address and size in bytes of the (de)allocated block it seems logically correct to differ for 32 and 64 bit architectures. In any case, note that the structure _MP_Record is 64-bit aligned on any architecture, thus there is no added padding. Records are not kept in memory but, within each wrapper they are serialized and added to a global buffer of fixed size. When the buffer reaches the limit size, it is flushed into the statistics file. The absence of padding enhances the serialization to be done optimally.

3.4 Output

The allocation data is written to a statistics file during the execution and has the following structure: a file header (Table. 4) and the serialized records. The file format of the statistics file is compatible with FOM-Tools so some of the fields, such as compression related ones, are present although they are currently unused for this version of MemProf.

Table 4. Structure if statistics file header

Field	Type	Role
key	HChar*	4 character identifier of the statistics file created by MemProf
ToolVersion	UInt	Number indicating the version of the tool
Compression	UInt	Compression ratio. Currently 0 because MemProf does not yet support compression
NumRecords	SizeT	Number of records stored in the file
MaxStacks	SizeT	Maximum depth of the captured stacktrace
BucketSize	SizeT	Size of the compressed buffer. Currently 0 because MemProf does not yet support compression
NumBuckets	SizeT	Number of compressed buffers stored in the file. Currently 0 because MemProf does not yet support compression
Pid	UInt	Process ID
StartTime	ULong	Process start time
StartTimeUTC	ULong	Process start time UTC
CompressionHeaderSize	ULong	Size of the compression header. Currently 0 because MemProf does not yet support compression
HeaderSize	ULong	Size of the header (including the variable size field - CmdLine)
CmdLength	SizeT	Size of CmdLine field

Following the file header, the statistics file contains the serialized records. Consider the following basic example:

```c
#include <stdio.h>
#include <stdlib.h>

int *a, *b;

void f1() {
    a = (int*) malloc(2 * sizeof(int));
}
void f2() { f1(); }
void f3() { f2(); }
void f4() { f3(); }
void f5() { f4(); }
void f6() { f5(); }
void f7() { f6(); }
void f8() {
    f7();
    b = (int*) malloc (5 * sizeof(int))
}
void f9() { f8(); }

void main () {

    f9();

    free(a);
    free(b);
}
```

Figure 4. Basic C program with 4 records

The program shown in Figure 4 contains 2 allocations and 2 deallocations, therefore 4 records. The corresponding records MemProf creates based on this program:

```
0x051fa040 8   0 11  1 0       107695  114958:   0 1 2 3 4 5 6 7 8 9 10

0x051fa090 20  0 4   1 518081  519268  520944:   0 11 9 10

0x051fa040 8   5 6   1 954447  955285  957171:   12 13 14 15 16 17

0x051fa090 20  5 6   1 1096294 1097552 1099088:  12 18 14 15 16 17
```

The field order is block address, size in bytes, operation type, number of IPs from stacktrace, thread ID, timestamps for beginning / end of memory operation and end of stacktrace generation and the indexes of the IPs from the stacktrace.

Additionally, there is one auxiliary file containing the mapping between indexes and actual instruction pointers used to restore the allocation context. This is useful for example to determine the location in the code of a certain memory consuming allocation.

The naming scheme for the output files is default <PID>-stat.log and <PID>-hash.log but the prefix is customizable via a command line parameter.

4 Usage

The usage of MemProf follows the pattern of all other Valgrind tools:

valgrind --tool=memprof <client-name> [<client-parameters>]

There are three accepted command line parameters:

Parameter	Use	Default
--uncompressed-buf=<number>	size of the uncompressed output buffer	64000
--stacktrace-depth=<number>	depth of the captured stacktrace	30
--log-basename=<string>	basename of statistics filename and hash filename	<PID>-stat.log / <PID>-hash.log

5 Results

MemProf is currently under development and has been tested over a number of sample cases among which we can enumerate a few standard Linux programs: unzip, telnet, netstat, evince and one CERN application cl_forward. Based on MemProf's output, we can perform several analyses for the client program such as memory access times, object lifetimes, locality, we can create statistics about the thread consumption, build memory allocation patterns.

Following graphs are built using GNUplot and matplotlib based on MemProf's output resulted in cl_forward analysis:

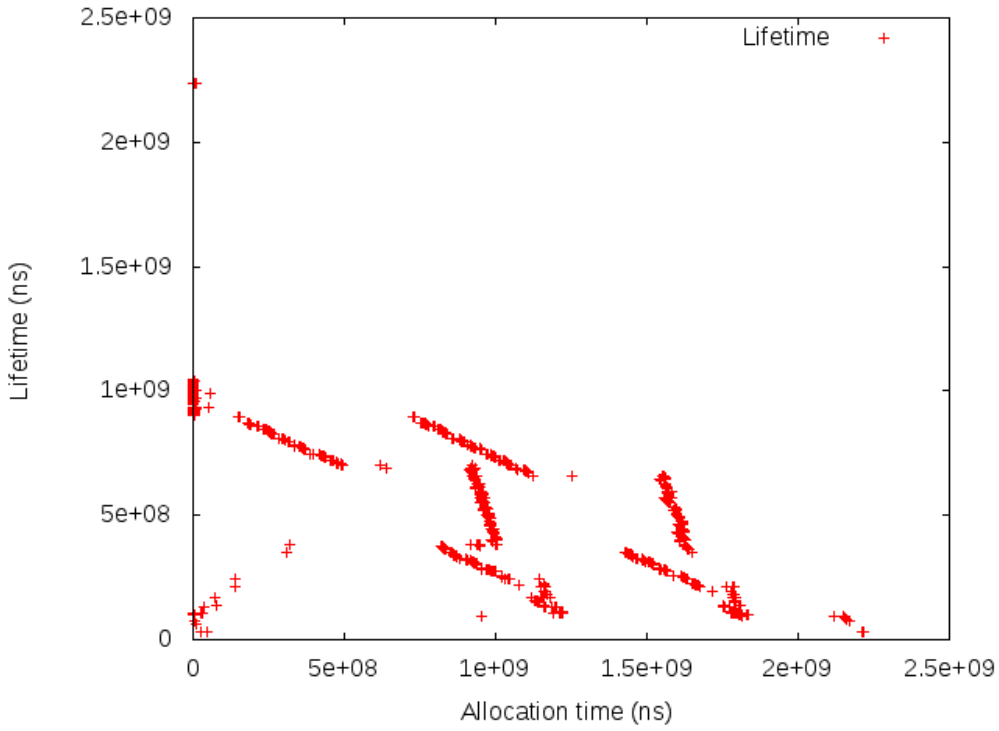

Figure 5. Object Lifetime

Figure 5 presents the relation between object lifetimes and allocation times in cl_forward. This kind of graph is a representation of the structure of the program and gives us hints about possible memory usage optimizations. For example, the two parallel lines visible around moment 1e+09 suggest that we have two series of objects allocated simultaneously but with different lifetime. We can deduce that this portion is mapped in the code to a loop instantiating objects with long lifetime and creating short lifetime auxiliary ones.

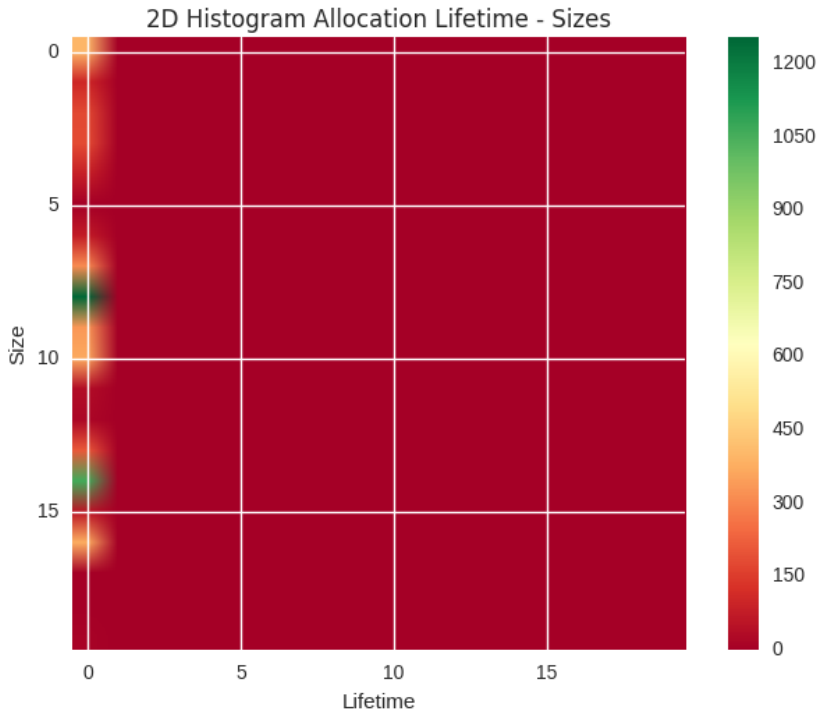

Figure 6. 2D Histogram of Allocation Lifetime vs. Size

Fig. 6 represents a 2D histogram between the object lifetimes and their sizes. This kind of graph offers a statistic of the number of objects for each combination of values <allocation lifetime, size>. We can deduce that most of the objects have a short lifetime and their sizes vary around two main values highlighted by the green dots. The graph is auto-scaled so there are few objects with longer lifetime but they are not visible in the plot.

The graphs resulted from testing other applications, such as standard Linux programs, are shown in Fig. 7. The test cases were the following: extracting a ~ 400 MB archive with unzip, performing a GET request for a resource on a web server via telnet, and printing network connections and interface statistics with netstat. For unzip, we observe that there are a series of object allocated at the beginning of the program (allocation time 0) that have the biggest lifetime which represent general auxiliary objects in the extraction algorithm and then a series of objects with very short life time corresponding to each extracted element. Similarly, telnet has a set of objects with longer lifetime holding the details of the connection and then short-lifetime objects are allocated for data exchange. On the other hand, as a network interrogation tool, netstat uses mostly short-lifetime objects.

For further testing or development, MemProf's code is public and available at the following address:

https://gitlab.cern.ch/cmoraru/MemProf

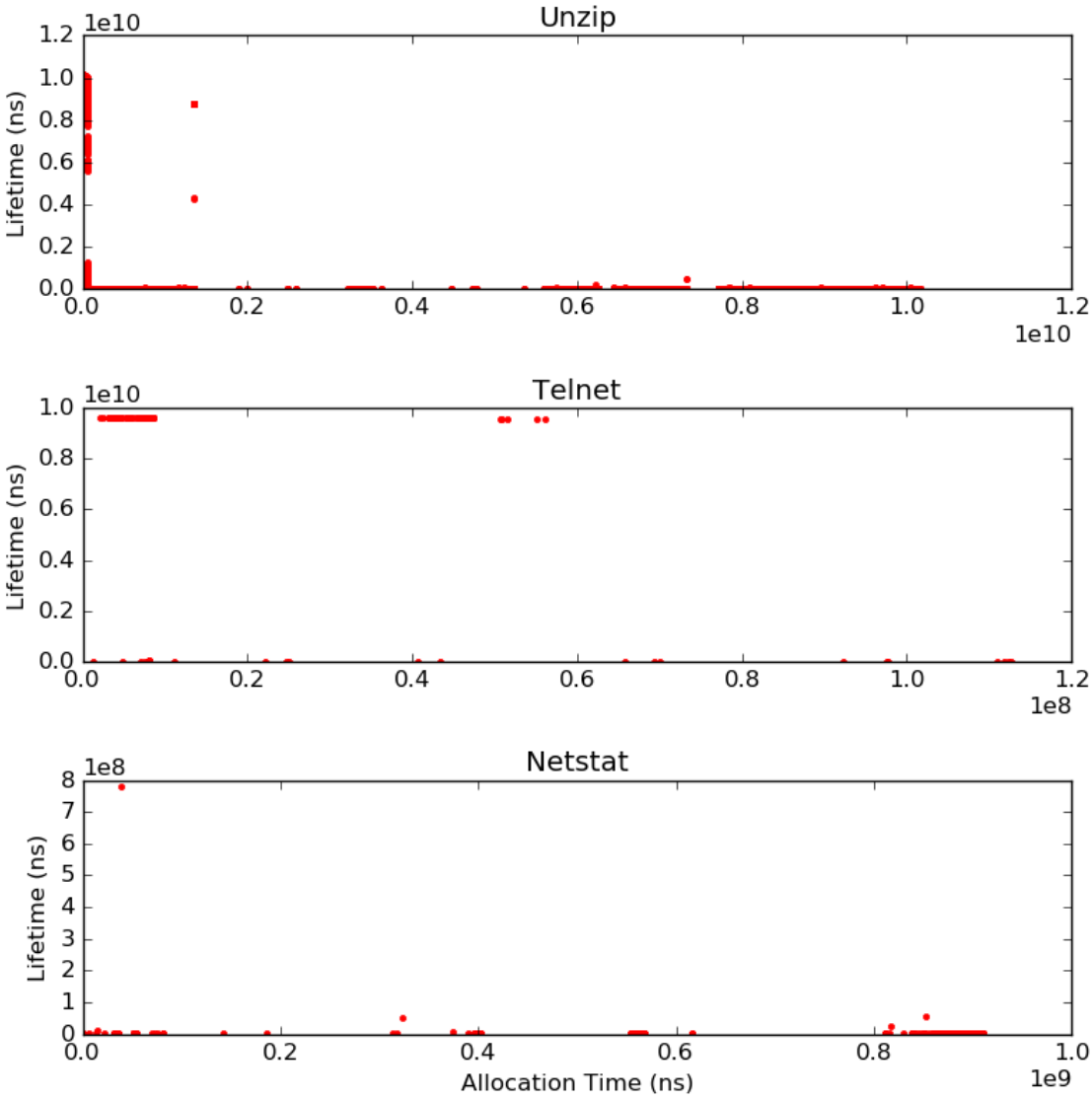

Figure 7. Results of testing standard Linux programs with MemProf

6 Further Work

Collecting memory allocation meta-information allows us to analyse many aspects of the memory usage of a program but the drawback is that the created statistics file is quite large, even for common applications. Therefore, the first step in the further work is adding support for compressing the output data with an algorithm such as BZIP2 / GZIP / LZO. Secondly, MemProf is currently able to track memory allocations but Valgrind's capabilities permit the extension to more fine-grained analysis. Tracking memory accesses would provide a deeper understanding of the memory usage by identifying objects that remain in memory longer than they are needed. The Valgrind framework offers the possibility of tracking read / write operations so this feature will be available in a next MemProf version.

MemProf is an easy-to-use Valgrind tool for memory allocation profiling, suitable not only for the software within CERN experiments but also for any data-intensive real-world applications. Due to its high applicability, we hope to integrate it into the public Valgrind repository to be available for larger use.

7 References

[1] Openlab summer student lecture 2016 - DAQ - Filtering Data from 1 PB/s to 600 MB/s - http://cds.cern.ch/record/2200792?ln=en

[2] FOM-Tools Wiki https://gitlab.cern.ch/fom/FOM-tools/wikis/home

[3] Cgroups wiki - https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

[4] Detection of Unused Memory and Memory Access Patterns, Nathalie Rauschmayr, Sami Kama https://twiki.cern.ch/twiki/pub/ITSDC/ProfAndOptExperimentsApps/Atlas_FSTF.pdf

[5] Pagemap official documentation: https://www.kernel.org/doc/Documentation/vm/pagemap.txt

[6] Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, Nicholas Nethercote, Julian Seward, 2007 http://valgrind.org/docs/valgrind2007.pdf