MALT: A Malloc Tracker

Sébastien Valat
Exascale Computing Research
Versailles, France
svalat@exascale-computing.eu

Andres S. Charif-Rubial
Exascale Computing Research
Versailles, France
achar@exascale-computing.eu

William Jalby
Université de Versailles Saint-Quentin
Versailles, France
william.jalby@uvsq.fr

Abstract

At the beginning of computer science memory management was a big issue with applications requiring to fit in the small amount of available memory (close to a few kilobytes). Hardware evolution has made this resource cheap for the past few years with now a few hundred gigabytes. But the current evolution tends to make some issues come back. The memory available tends not to follow the increasing number of cores making the memory resource per thread rare again. We also encounter new issues with the requirement to manage a bigger space with many more allocated objects. It increases the probability of memory leaks and the probability of memory management performance issues. Hence, with MALT we provide a tool to track the memory allocated by an application. We then map the extracted metrics onto the source code, just like kcachegrind does with valgrind for the CPU performance. Compared to most available tools, MALT can also be used to track potential performance losses due to bad allocation patterns (too many allocations, small allocations, recycling large allocations, short-lived allocations...) thanks to the various metrics it exposes to the user. This paper will detail the metrics extracted by MALT and how we present them to the user thanks to a nice web based graphical interface which is missing with most of the available Linux tools.

CCS Concepts • **Software and its engineering → Allocation / deallocation strategies**;

Keywords memory,allocation,management,tool,profiling

ACM Reference Format:
Sébastien Valat, Andres S. Charif-Rubial, and William Jalby. 2017. MALT: A Malloc Tracker. In *Proceedings of 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141865.3141867

SEPS'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5517-9/17/10…$15.00
https://doi.org/10.1145/3141865.3141867

1 Introduction

It is known for a long time, the current evolution of computer architectures tends to make memory a bottleneck [27]. Memory is a performance bottleneck when considering the memory accesses, for example by looking on the cache optimized algorithms. But, as the compute power tends to increase faster than the available memory [2], memory is now also an issue for the availability of the resource itself. This is even true when looking on the available memory per core or per thread. The memory space to manage also increased a lot with now hundred gigabytes available. This opens the door for management performance issues due to more intensive usage of the memory allocator.

Considering this context, developers might be interested in getting tools to profile the memory behavior of their code. They might ask the two following questions:

- It appears I used too much memory. Where does it go? Which portion of my code requested it?
- Memory allocations can hurt performance. Are there some bad allocation patterns (over allocations, lots of small allocations, short-lived allocations...)?

But, it occurred to us that the Linux environment lacks such a tool with a simple and attractive graphical interface. Hence, we describe here our work on a memory profiling tool to track memory allocations by providing memory profiles just like valgrind [14] / kcachgrind [26] does for performance. Our approach aimed at mapping the memory allocations onto the source code and call tree to help developers pinpoint the interesting locus quickly and fix the code. We will also take into account the temporal aspect to capture the dynamic aspect of the application life cycle.

This paper will first introduce the basics of memory management to give the main concepts used into our tool. Then we will list the already existing products to fulfill similar work and from which we took inspiration. We will follow up by detailing the concepts used in the tool, looking on the back-end side, about instrumentation and at the GUI (Graphical User Interface) side. We then present some real use cases showing the value of the tool and some performance results including the overhead of the tool.

2 About Memory Management

From the Operating System (OS) side, memory management is based on the paging system [19] [8]. The virtual and physical address spaces are described as pages of 4 KB by building a translation table managed by the Operating System (OS)

and the processor. This imposes a constraint on user space applications. They need to request memory segments with size multiples of the page size. To solve the size alignment issue, the user space runtime library (libc) provides some functions to allocate such aligned memory segments and split them into smaller ones matching the user needs.

From the API point of view, in C, C++ and Fortran, this memory splitting management relies on the malloc() and free() functions. Those two functions respectively allow the caller to dynamically allocate a segment of a certain size and deallocate it when not used anymore. They are extended in C by a couple of other specific functions. The realloc() function is used to resize a previously allocated segment. The mem_align() and posix_memalign() functions permit the caller to force some specific alignments on the allocated addresses. The other functions are variants of the previous ones and are outside of the scope of this article, although they are tracked by our tool.

Segments allocated by the OS are initially only virtually defined and do not contain any physical pages. This is called lazy page allocation. The first access (also called first touch) in the pages of a new allocated segment will generate page faults. It notifies the operating system of the first access and matches the virtual page with a physical one. When tracking the memory usage of an application we need to take care of the physical and virtual memory usage separately as they can differ.

Regarding the performance, the lazy allocation strategy can slow down the application if it is used too often by allocating and freeing large segments many time. It is even more true when considering that each page has to be cleared with zeros before being mapped. This can induce large overhead for the application as discussed in [25] by looking on memory management overhead on HPC applications.

3 Related Work

Of course we can find some existing tools in the Linux ecosystem to answer the two questions from section 1. We can distinguish two kinds of tools. The memory checkers are used to validate usage of memory allocation functions and memory accesses. In this first category, we mainly cite the memcheck tool [14] based on valgrind [13], drmemory [3], electricfence [17] and AdressSanitizer [21] from LLVM. They provide information about memory misusage, memory leaks and access to invalid memory regions like freed segments. Those tools mainly track the memory allocations and accesses to generate a final report summarizing the detected errors. Notice that all of them provide console text outputs which might make it hard to read and filter when getting large amounts of reported errors. To solve these issues some GUIs are available to handle the output of valgrind memcheck but they are quite simple. Valgrind has the specificity to run by emulating the CPU hence playing each instruction. It permits to get deep analysis of instructions but implies a large overhead as we will see later.

On the other side, we found tools to track the temporal and spacial (call sites) aspect of the memory consumption. Most of them use snapshots of the memory usage at some points and report the origin of the allocation in the source code. It is, the case with massif [15] from valgrind and Google heap profiler [9]. We consider a snapshot as building a list of all the allocated segments at a given time with their origin (call stack or call site). Then the tool can give each snapshot as output as Google heap profiler or print all snapshots on a timeline with the top X functions like the valgrind massif GUI. Similarly, we can cite igprof [24] developed by CERN. It uses a similar instrumentation method to ours but with fewer metrics and no GUI. IgProf provides a mapping of allocation amount mapped onto the call sites. Like us, Google heap profiler and igprof proceed by preloading a library to override the malloc symbols and capture calls from the application to build their metrics.

We then can find tools which look deeper in the allocation structure and access patterns like Hound [16] providing leaks and bloat detection. Similarly we found FOM [18] developed by CERN which tracks all the memory accesses of your memory allocation and provides hints about chunks which can be freed more quickly because they haven't been used in a long time. Those two tools rely on usage of mprotect() to lock the pages and periodically track accesses. FOM also uses a trace approach by dumping all the allocations into a database for offline analysis. But notice that the database can become huge when running applications making billions allocations.

Concerning explicitly parallel tools, one can look into the HPC community and find the memory profiler embedded into TAU [22], which is also close to our approach (at least for the back end). It uses a preload approach and a global profile mapped on functions. But it provides a limited amount of memory metrics compared to MALT, mainly seeking the amount of allocated memory and the number of calls to memory management functions. As with our solution, it prints global statistics for each called function. Compared to us, it has the great improvement of mapping the allocations to the multiple MPI tasks in use. This point is currently outside of the scope of our tool. Similarly we found memP [5] targeting the functions allocating memory at the memory peak for every MPI task. This tool is compatible with a generic old TCL/TK GUI which does not add a real value compared to the text output.

Looking at the commercial tools, we can find Insure++ from Parasoft and Purify++ [10] which are commercial products available on Windows and Red Hat. Those two tools provide a complete and detailed GUI. We also found the tool embedded into Visual Studio Ultimate Edition. This last one is close to what we want to provide with the MALT GUI with source annotations and per call site measurements. Those tools provide memory usage over time and per function

plus some other information to dig deeper into the analysis. But as said they are not free and not portable to all Linux distributions.

4 Design

We previously detailed the memory management method used by C/C++ and Fortran languages and some available tools to track memory usage. We will now detail our own approach to provide the user with a profiling tool to track memory allocations and map them within the source code in the web-based GUI. We will first define our objectives and our global approach. Then, we will detail some selected points of our implementation showing the specificity of MALT compared to the available tools. We will then discuss shortly some of the interesting points of our graphical interface. In addition to the original front end, our main improvement is to take the lessons from the available tools and push forward the analysis by providing more metrics to the user, mainly adding hints on the performance issues.

4.1 MALT, Definition

Our tool is built upon three major concepts:

- It captures memory allocation by wrapping the standard C memory functions (malloc, calloc, free, etc.) via the well-known LD_PRELOAD Unix approach It works just like Google heap profiler, igprof, FOM and TAU.
- Inside the memory management function, MALT captures the call stack to build a mapping onto source code.
- MALT builds a memory profile by mapping memory allocation metrics on a sampled timeline and separately on the call graph of the application. This dual mapping is a benefit of MALT over the other tools which do either one or the other.

4.2 Interesting Metrics

Our main objective is to report information about memory consumption and potential performance impact. We want to know where the memory is allocated and how. Targeting the "how" is a major improvement compared to the available tools. From our understanding it seems interesting to provide the following metrics to users:

- Memory allocated by each call site at the application memory peak time. This metric provides information about what to improve in the application to reduce the memory footprint. It is what we want to search if an application failed to run due to memory exhaustion or swapping.
- Minimal, average, and maximal chunk size allocated by each call site. Allocator developers know that each allocation requires a small overhead to store allocator headers for each chunk. Thanks to these metrics the users can quickly point to call sites using too small allocations and merge them if possible. It also allows for finding large allocations and give advice if some can be shrunken. None of the existing tools provide such a metric at least for each call stacks.
- The allocation count per call site allows pointing out potential performance impact if the application generates too many allocations. This metric is available in tools like TAU. It can also be extracted from performance evaluation tools in a generic way like with callgrind from valgrind.
- The cumulated memory allocated by each call site targets the same objective, but also provides information about the potential cost of the OS due to lazy page allocation. It also points out memory intensive functions as it is partly correlated to the memory consumption. None of the existing tools provide this metric.
- Minimal, average and maximal lifetime of chunks allocated by each call sites permits characterization of the usage of the allocated chunks. It can be interesting to track short lived allocations to improve the application performance. It can also help to track potential fragmentation issues if a function mixes short and long-lived chunks of similar size. This metric is not present in any existing tools.
- We can create a metric by reusing the cumulated and peak (local, not global) memory of each call site by computing their recycling ratio. It provides information about how many times the function reallocates its memory. Ideally, for performance, this amount might be one with memory allocated at the program start and freed at the program termination. This metric is not provided by any existing tools.
- Requested memory over time to be compared to the physical and virtual global memory. It can help to detect fragmentation issues if it diverges as shown later by the example of section 5.2. It is what is roughly provided by tools like massif (only for virtual memory).

Most of those metrics are absent from the open-source tools listed in the related work section. Hence, MALT might provide an improvement in the memory tracking field. MALT also tracks free and realloc operations with similar metrics. This information is absent from the available memory tools which just ignore the statistics about those two functions.

4.3 Flat Profiles Versus Snapshots

Some of the available memory tools use a snapshot approach to provide the temporal and detailed view of the memory used by the application. It has the negative effect of ignoring the operations performed between two snapshots. If the memory peak appears between two snapshots (as in figure 1) the user will not see it in the data exported by the tool. It is partly fixed in valgrind/massif by some tricks like taking

snapshots only on deallocation and using a peak inaccuracy of 1%. But we observed that it still misses the secondary peaks if they are short. Similarly, if a function does many cycles of allocation/deallocation between two snapshots, the user might not see them and will ignore their performance impact. It is also hard to decide when to take the snapshot without knowing the final execution time of the application. Hence, it is necessary to dynamically rescale the time steps. Or, one can rely, as Google heap profiler does, on a total allocated memory threshold. But the choice of the default value is arbitrary and might need to be changed by the user.

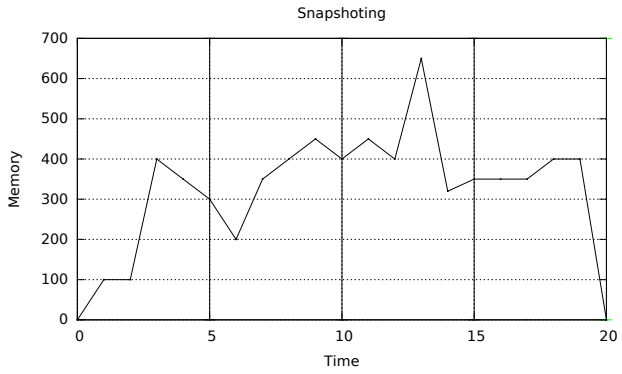

Figure 1. Here is an example of snapshoting every 5 steps. In this case we missed the global peak at step 13 and possibly all short life allocations in-between snapshots.

In our approach, like in TAU, the flat mapping onto the call stack ensures that we take into account all the memory operations and report them to the user in the output profile. But it has the disadvantage of losing the temporal aspect. So, as a complement, we also project the global memory consumption metrics onto the execution timeline (only globally, not for each call site). We consider that the detailed temporal aspect of each call site can be reconstructed from a trace based approach in a second step for a deeper analysis. Thanks to this flat mapping approach, we avoid the complications of tracing methods for large applications making billions of allocations. Hence it limits the profile file size to a ratio linked to the call tree size instead of the absolute number of allocations (total for traces). It is useful when the application makes billions of allocations.

We can compare the MALT output file size aspect of the tracing approach and the main profiling methods of MALT by looking at Figure 3 in section 4.6. Those values were measured by running the tool on various applications making around ten million allocations and testing different stack tree storage strategies.

4.4 Memory Peak

We explained that we use a temporal mapping of the global memory used by the program. As we cannot save all of the points over time (which is a trace), we need to apply a sampling approach by capturing a finite number of states for the given metric over time. As we want to ensure we capture the peaks, we do not sample the value itself but sample its maximal value on each time interval. It also has benefits of easing the dynamic resampling to keep a fixed number of points while considering an unknown time window. Effectively we keep the max value of the two merged points and do not have to decide "randomly" which one to keep. Thanks to this, we ensure the capture of all the peaks (primary and secondary), even if we lose the time precision associated with the value. Notice that we could also track the minimal value as a complement to detect oscillating patterns between captured points. This approach is not used by the existing tools listed in the related work section.

The other point is the capture of the memory consumption of each call site at the global application memory peak. To extract this information we ideally need to make a complete snapshot of the allocated chunks at peak time. Since we do not know in advance when this peak will appear, we have to snapshot the memory consumption for each call site on each new maximum memory consumption.

This approach implies a large number of snapshots at start time before reaching the final peak. It will effectively capture all of the points of the first phase of memory allocation. Taking a full snapshot in this condition can induce unacceptable overhead to extract the metric. Opposite to what does massif, the problem can be solved by considering a lazy update of the peak counter attached to each call site by following the given approach and implemented in Figure 2:

- Define a global peak ID which is increased each time we detect an increase of the total memory used by the application.
- Each call site stores its current memory consumption, last peak consumption, and the last peak ID to which it relates.
- On memory operation, a global peak ID larger than the one from the current call site implies an update of the last peak consumption. In practice it's a copy of the current consumption (before taking into account the current operation) onto the local peak value.
- A subtle point of the implementation is to update the peak ID after updating the call site to prevent capturing incoherent values depending on whether or not the current step is a new peak.

Thanks to this approach, we do not need to update the whole call tree every time we reach a new memory peak. Then, we just ensure a global flush at the end before dumping the profile into the output file.

4.5 Stack Reconstruction

In MALT we provide stack tracking through different mechanisms. By default we use the glibc backtrace() function

```
gblMem ← 0
gblPeakMem ← 0
gblPeakId ← 0
procedure onMallocOrFree(stack,deltaMem)
    local ← getStackInfos(stack)
    if gblMem > gblPeakMem then
        gblPeakId ← gblPeakId + 1
        gblPeakMem ← gblMem
    end if
    gblMem ← gblMem + deltaMem
    if gblPeakId > local.peakId then
        local.peakId ← gblPeakId
        local.peakMem ← local.mem
    end if
    local.mem ← local.mem + deltaMem
end procedure
```

Figure 2. Algorithm to lazily update the local memory consumption on global peak.

called from inside the allocation function. This method has the advantage of working without external dependencies and captures the whole call stack, including libraries functions. But this approach has the inconvenience of inducing a large overhead when applied to applications generating millions or billions allocations. It also leads to crashes with some libraries like Intel OpenMP or Intel MPI, depending on the internal stack optimizations they use. A complementary and more stable approach relies on libunwind to replace the glibc backtrace. This can help to handle situations where backtrace crashes. It is the approach permitting igprof to sustain a decent performance. We currently retain the libunwind method only by explicit demand from the user.

The other approach consists of instrumenting the program to be notified on each function enter/exit point. This way the tool can dynamically reconstruct the call stack[4] without the inconvenience of walking through the stack. It can be done by using source instrumentation thanks to the -finstrument-function option of gcc/icc or similar for other compilers. This approach requires a recompilation of the program and will only permit to see the functions defined by the program itself. The system libraries we do not recompile are skipped and hidden in the profile. Similarly it might be interesting but not yet done, to look for integration of the work done in HPC profiling tools like Tau [1] and ScoreP [12]. They provide well-integrated stack instrumentation methods and also provide management of MPI based distributed applications at the same time.

The final approach uses binary instrumentation to insert probes on each function entry/exit point (similarly to the source instrumentation). This method can be applied by using tools like maqao [6] or pintool [11] and can also take into account the dynamic libraries by instrumenting them as it does not require the sources anymore.

In our implementation we provide support for those three major approaches. Backtrace is taken by default because it is simpler and more generic for the user. We provide some basic integration into maqao and pintool but they are currently not as well implemented than the backtrace and compiler methods.

A last approach which has not yet been studied yet would be to include the MALT library into a valgrind plugin. It offers a nice way to handle the call-stack tracking but might induce a large overhead especially on multithreaded applications.

4.6 Stack Tree Representation

As explained, MALT supports different instrumentation methods to get access to the call stack. It has some impact on the internal representation. In the backtrace mode, we obtain the stack as an array of pointers going from the leaf to the root. To handle it efficiently we hash this array with a simple function and build a red-black tree using the std::map. To remove the ambiguity between same hashes we build the key applying the following operations in given order: hash, size, full-stack comparison. Hence we get a balanced tree to quickly access the stack independently of the structure of the real call tree. It also opens the door to easily implement some caching method (not yet done).

When using the source or binary instrumentation mode, we reconstruct the call stack also as an array of pointers. We use it in the same way as the backtrace mode when reaching a memory management operation. Note that it avoids reconstructing the full call tree. MALT only stores the entries linked to memory management calls.

The stacks are directly dumped in this key/value format into the JSON output file. We are currently using NodeJS to host the graphical interface and load this JSON file for data extraction. But it appears that we can reach one NodeJS limit as it cannot load the profile file when reaching a size of around 600 MB. This size was reached with the CERN LHCb online software stack [23] which is 5 MLoc of C++ code.

We solved the issue by providing a tree based representation for storage which is more efficient in some cases. We also applied an algorithm which optionally optimize the call stacks by removing the recursive call loops which are uninteresting for the user. It saves a lot of space in some applications. For the LHCb case, it divided the profile file size by a factor 10 as shown in the Figure 3.

4.7 Tracking Global Variables And TLS

Fortran simulation codes often make heavy usage of global variables, which can lead to memory consumption issues. As they are not allocated through the dynamic allocator, they are ignored by most of the available tools. But, they can in some cases, represent the major part of the memory consumption.

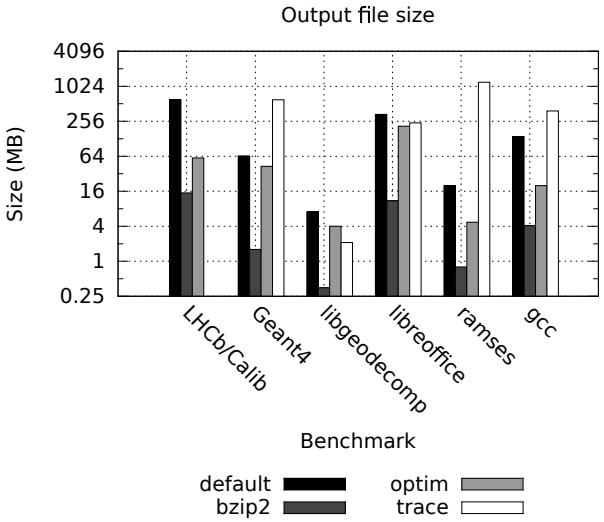

Figure 3. Here are some examples of output file size providing the default size, compressed with bzip2, optimized with loop suppression and stack tree representation and the optional trace file. The size axis uses a logarithmic scale.

As we want to present a global view of application memory usage to the developer, we also added the necessary code to extract and display the memory used by the global variables. It is done by analyzing the ELF file information of each binary module loaded by the program. We used libelf for this task.

Current applications start to become multi-threaded so the global variables now can be stored into TLS (Thread Local Storage) in C-based languages. This kind of global variable is global to a thread but duplicated for local use in each thread. We also take into account this case by checking the respective section of the ELF file and multiplying the related sizes by the peak thread amount plus one for the original copy. None of the tools we listed provide such an analysis.

4.8 Memory Perturbance

MALT has a large performance overhead. Hence, the timings are surely biased. If we do not take care, such instrumentation might also change the memory layout of the application by interleaving the application allocation with the one of the tool. In order to avoid this effect MALT uses its own internal allocator not to mix its allocation with the application one's.

This allocator also ensures to make the memory segment physically mapped so we can ensure that the virtual and physical memory used by the allocator match. It permits to request the tool memory usage and remove it from the global physical and virtual memory tracking over time. This is interesting because MALT reports the physical memory used by the program by making a request to the operating system which returns the global consumption including the memory used by MALT.

4.9 The Graphical Interface

Our first idea for this tool was to apply the valgrind and kcachegrind approach to the memory concepts. Hence, we initially made our tool able to produce files compatible with kcachgrind. But this approach quickly reveals several limitations:

- Kcachgrind displays metrics as raw numbers. When measuring cycles or cache misses we are more interested in the relative values between call sites than the absolute ones. But this is not true for the memory, as we want to compare the consumption of functions relatively to the memory available on the system or to our expectation. To this end, we need to provide a better way to represent the metrics by using a human readable approach with units (KB, MB, GB, etc.), as the user cannot easily read long numbers. So we need the GUI make the conversions itself. Notice that this is something absent from all the available profiling tools.
- Kcachgrind is built for cumulative metrics, meaning metrics that can be summed up to build the inclusive / exclusive costs and compare them to the total cost (sum). With memory it is not true for all of the metrics we targeted. For example, the local memory peaks might not appear at the same time, so, cannot be summed up. It's similar to the chunk lifetime or chunk sizes. The operation to reduce those values is not the addition, but a minimum or maximum.
- There is no way to provide temporal information or to display the global statistics we wanted to provide (eg. global and TLS variables).

Of course kcachegrind can be patched to support those semantics and it might be interesting to do it. But, we also wanted to exploit different and innovative approach. Hence, we decided to build a new web-based interface by using up-to-date web technologies like D3JS for the charts. If coupled with NodeJS, this approach has the advantage of being easy to run remotely and of using port forwarding through SSH to get the local view without too much latency on slow connections (as opposed to using the X forward approach). This might be a plus for usage in HPC field. It also provides the possibility for two remote users to connect to the same server to easily dig into the profile and discuss their desired optimization. In any case, if someone wants a fallback to the default X11 approach it is possible to mix NodeJS with the chromium browser into a single application thanks to the Electron framework. It is not yet implemented in MALT.

The tool produces an output in JSON format in order to be compliant with web technologies. This format also confers the benefit of exploiting a more reusable format.

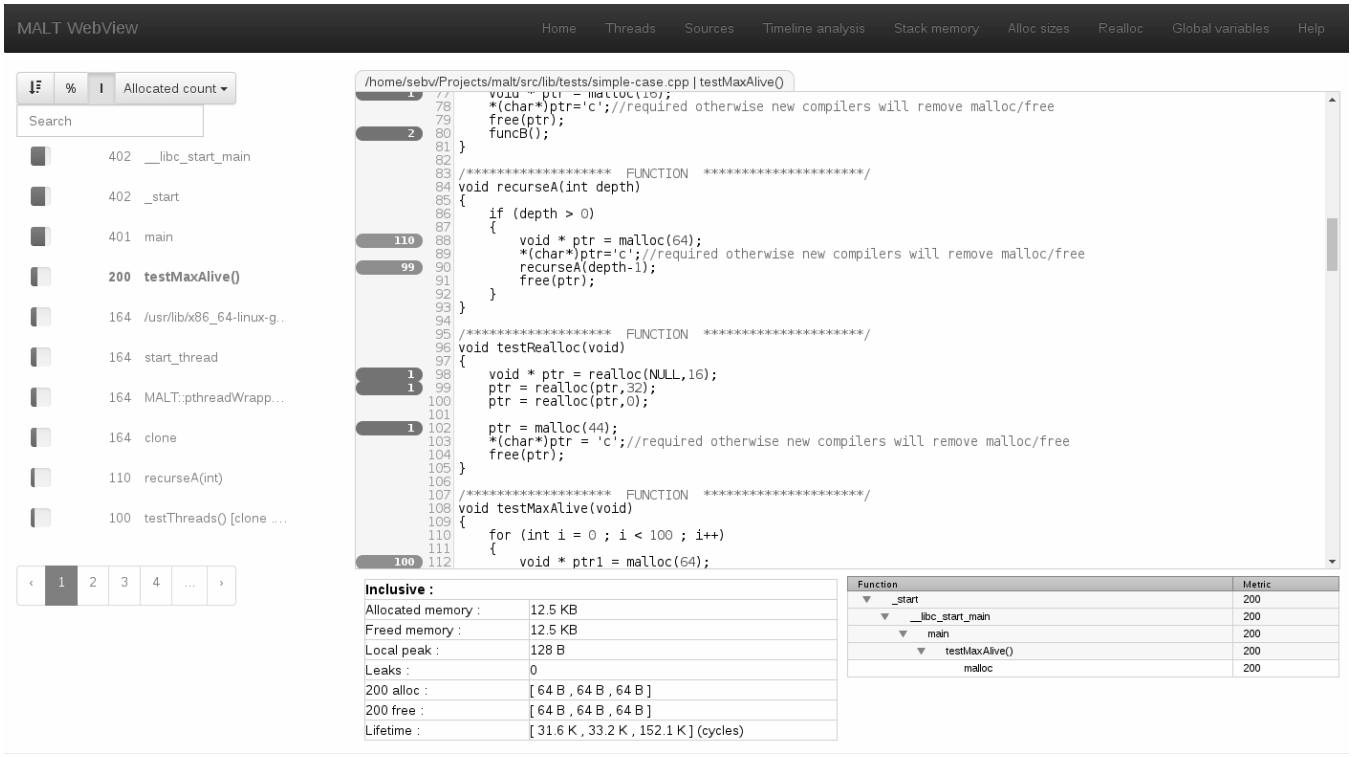

Figure 4. Source annotation view of MALT.

The drawback is a larger file size but it can be overcome by compression.

Our graphical interface currently provides:

- A summary of the execution and some global metrics (allocation amount, cumulated allocated memory, larger chunk size, recycling ratio...). It is presented as an array and can support some warnings if some values might show an issue like the CQA tool [7].
- Top 5 functions for the most interesting metrics (memory on peak, the amount of allocation, memory leaks, cumulated allocated memory).
- The source code annotation browser shown in Figure 4, similar to kcachegrind view. It shows on the left the list of functions annotated with the selected metric. On the right is the annotated source code. On the bottom is the annotated call tree leading to the selected function and some detailed metrics about the selected call site.
- Some time based metrics, mostly, the virtual, physical and requested memory. It also shows the allocation rate to detect possible performance issues, in bytes or requests per second.
- Distribution of used sizes.
- Scatter plot showing the distribution of allocation sizes over time and lifetime over size. This can help to understand the memory performance profile of the application.
- Size jumps (origin size to final size) used into realloc and distribution over sizes.
- Global variables and TLS memory.
- For each thread, the amount of memory management calls considering the total time and the number of calls.

5 Use Cases

We tested MALT on some numerical simulations like Yales2 from Coria, AVBP and Dassault miniapp. With those applications we are able to show some interesting use cases of the tool as described here.

5.1 YALES2: Unexpected Allocations

Yales2 is an HPC application solving chemical equation for flaming fluids. It is written in Fortran that uses the allocatable keyword to manage dynamic arrays. Thanks to MALT we observed that the version compiled with gfortran generates many more allocations than the one compiled with ifort. We observed that each loop using allocatable arrays generates memory allocations/deallocations that weren't expected by the user from the given code. Searching deeper on this issue showed us that ifort and gfortran use copies of the arrays to ensure some access properties (vectorization). Ifort uses the stack for arrays smaller than a configurable limit, which is infinite by default, and gfortran uses dynamic allocation by default. Of course an expert Fortran developer might know this, but most users (like us) might not. Thanks to MALT

we identified the case in a couple of minutes looking at the annotated sources. The example is shown in figure 5.

Figure 5. Example of unexpected memory allocation generated by the compiler (gfortran) itself.

5.2 Dassault Miniapp: Fragmentation

The Dassault miniapp is a research project from Loic Thebault and Eric Petit to explore modern solutions on a simulation from Dassault aviation. While using MALT on this application we observed, in the temporal charts, a divergence between the physical and the requested memory, as shown in the Figure 6. We observed that, at some point, the requested memory drop down when the application finishes a step, but not the physical memory. At this time position, the application frees temporary memory. But, it also allocated some long-lived chunks at the same time. They mixed up with the previous one. Hence, the allocator cannot return the memory to the OS due to fragmentation.

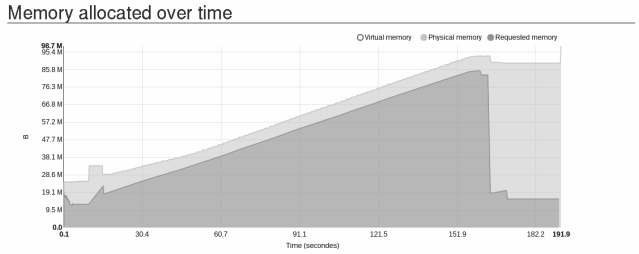

Figure 6. Example of fragmentation detection on time charts by looking on the divergence of requested and physical (or virtual) memory. The drop of requested memory indicates a massive free operation. But the missing drop of physical memory indicates a missing impact on the memory consumption (meaning fragmentation).

Thanks to the tool, we are able to observe the phenomenon on the charts and dig into the source (via the source annotation and call sites allocation metrics, using the allocation amount and lifetime) to find the allocation points. After a couple of minutes we were able to determine which useless short life allocations might be removed or reduced to improve the situation. In practice it leaded to a great improvement in the memory consumption of the application, making it possible to run on the smaller memory of the current Xeon Phi (whereas, before, this was not possible to run). It also improved the overall performance of the application as it went to swap for large meshes and never shrunk down after the initialization phase due to fragmentation. Thanks to the changes performed after MALT profiling, it does not swap anymore.

5.3 AVBP and MAQAO: Realloc

While using MALT on applications like AVBP and MAQAO we successfully detected an intensive allocation pattern while reading the temporal chart about allocation rates (Fig. 7). Thanks to the annotated sources, by looking at the allocations amount, we quickly found the source of the allocation. We observed a large reallocation pattern consisting of allocating an array of a certain size and reallocating it later to another one to make it grow. In Fortran this pattern lead to a new allocation, copy, and deallocation of the old segment. This generates far more memory operations than it needs with the C realloc approach. The solution in this application was to make it unnecessary for the growing of the array to recur so quickly. This can be done by growing more the array in advance thinking of the next step to not do it for every step.

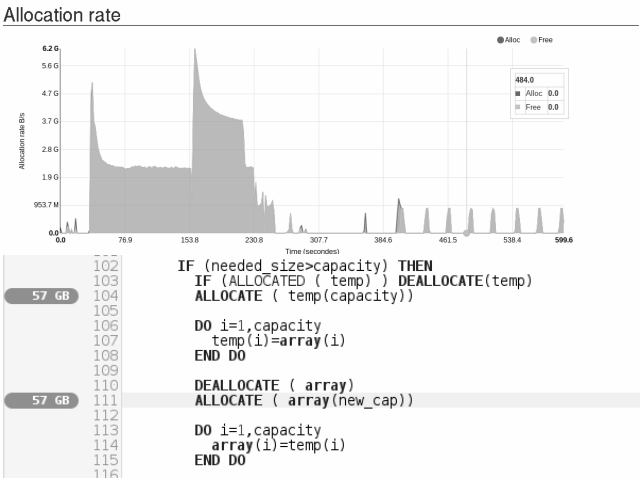

Figure 7. Example of heavy usage of realloc implemented as alloc/copy/free in Fortran. On top the allocate rates over time with up to 6 GB/s which is likely to to produce a slow down caused by the operating system. On bottom the annotated sources using the allocation cumulated size metric.

Similarly, in MAQAO we observed a reallocation pattern. In this particular case, by reading the MALT annotations, the developers saw they can totally remove those allocations by slightly changing the algorithm.

5.4 Summary

Those examples show that it can be helpful to provide a simple way to measure and summarize the memory allocations performed by a program and that developers can successfully improve their code if we provide the information they missed. Most of the time there is what we think about our

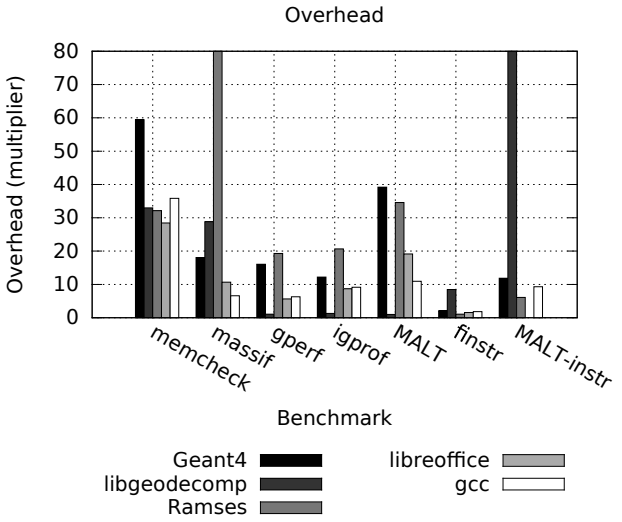

Figure 8. Example of overhead of MALT for various applications using the default backtrace mode or the source instrumentation mode. The overhead is obtained by taking the instrumented execution time divided by the standard execution time. Caution, the chart was cut to stay readable, but the libgeodecomp entry in finstr mode reached a factor 126, but it is only 20 compared to the default finstr mode.

code and what it is. Memory tools are needed to help to fill this knowledge gap about memory usage in our applications.

6 Performance

The overhead of MALT mostly comes from the call stack tracking, which by default is due to the backtrace() method. Hence, the overhead is directly linked to the amount of memory allocations performed by the application. We provide some overhead measurement in Figure 8 for various applications. The results included in figure 8 contain:

- Geant4 which is a simulation framework for particle physic, using the wholeNuclearDNA example. This application does close to 20 million allocations with big C++ call stacks and using threads. It shows the large overhead of MALT.
- libgeodecomp [20] which is a MPI stencil simulation framework using a reasonable (100K) amount of allocation. It shows that MALT can have a quasi-null impact on the performance. But, it points some limits of the source instrumentation approach with C++ codes heavily relying on inline properties which induce a large overhead.
- Ramses is a numerical AMR simulation written in fortran and generating close to 40 million allocations per process for the tested case. We used 16 MPI processes for the test.
- libreoffice also provides a large C++ code base making 15 million allocations.
- GCC is a large C code base using many small allocations. In the tested case, by compiling a large template based C++ file it generated more than a million allocations.

MALT can reach an overhead close to half of the valgrind overhead with the default backtrace method with up to a factor 40. But it also showed a quasi-null impact on applications using a low amount of allocation. For large amount of allocations the source instrumentation approach might be preferred and except for one corner case provides overhead bellow 10x. Hence getting those dual approaches we are able to maintain an overhead bellow 12x in any of the tested cases by choosing the right method.

7 Contribution

The available tools are helpful but most of the ones available on Linux lack a compelling GUI which can be helpful for the developer. With a GUI he can work similarly to performance evaluation with kcachegrind and Visual Studio. This way the developer can figure out where the allocations take place and relate them to the source code. It is also useful when the tool starts providing a large number of outputs which mostly occur in large applications. With a GUI it is easier to filter and sort the data within the sub-component of the code. Hence, not being flooded by too many raw outputs. This is for example an issue when using google heap profiler.

Additionally, the snapshot approach in use by many Linux tools has its own limitations. In this approach the tool might miss the memory allocation peak between snapshots. It also does not show the performance impact of functions allocating and de-allocating large number of short-lived buffers between two snapshots. In MALT we generate a flat profile on the call stack, removing the timing information to capture those events. Also, some of the metrics provided by MALT target the performance aspect of memory management, a point not touched by any of the available tools even the commercial ones. It is mostly done by looking at the allocated chunk properties like lifetime and size.

As explain in section 4.4 MALT use a special approach to capture the memory peak and make a snapshot at this position by using a lazy update method which is a different approach than the existing tools like valgrind massif and google heap profiler.

In order to answer the two initial questions from section 1, we decided to build a profiling tool by mapping memory profiles onto the two axes:

- Mapping of allocation statistics to call stacks (meaning source code and call tree).
- Mapping of memory consumption over time to show the dynamics of the code just like snapshots based tools do but without capturing the stack details.

Due to this dual approach, MALT tries to take the best parts of the two existing profiling worlds opposite to the existing tools which use either one or the other. It also tries to present the output in a useful way to the user in an attractive interface. In MALT we try to annotate the sources code and track the call stack with more metrics than the existing tools to provide a characterization of the allocations to the user. In that sense, MALT mostly aggregates in one tool the interesting features of the existing ones. We showed in section 5 that we successfully used MALT to find interesting allocation patterns in real applications. MALT also provides some metrics about the memory consumed by global variables and TLS which is a novelty compared to existing tools.

8 Conclusion

We discussed the lack of memory profiling tools on Linux, for the Open Source community. Hence, we proposed MALT which aimed at filling this gap by providing a tool which map the memory allocations on your source code in a human readable way within an advanced GUI. We clearly show the interest of the extended metric we provide compared to existing tools. We also showed the interest for a non-snapshot based approach by providing a flat mapping onto the call tree and time separately. Finally, we provided some real use cases where MALT permits to discover unexpected behavior of real C, C++ and Fortran applications. There is still rooms for improvement but we showed on those application that MALT can maintain its slowdown lower than 12x using various stack tracing approach.

For the future, MALT needs some work to improve its interface mainly to make it a more generic. Hence it might be reusable for other metrics as its core functions provide a nice way to build similar analysis tools for other fields (I/O...). It will also be interesting to embed the MALT library into tools like Valgrind and ScoreP. MALT can certainly be extended to provide some metrics about NUMA.

Acknowledgments

Thanks to the Exascale Computing Lab for financing this project.

References

[1] Robert Bell, Allen D. Malony, and Sameer Shende. 2003. *Euro-Par 2003, Austria.* Chapter ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis, 17–26.

[2] Keren Bergman, Shekhar Borkar, and al. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. (2008).

[3] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory *(CGO '11)*. IEEE Computer Society, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[4] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 76, 11 pages. https://doi.org/10.1145/2544137.2544164

[5] C. Chambreau. 2010. memP. (2010). http://memp.sourceforge.net/

[6] Andres Charif-Rubial, Denis Barthou, Cédric Valensi, Shende Sameer, Allen Malony, and William Jalby. 2013. MIL : A language to build program analysis tools through static binary instrumentation. In *High Performance Computing*. India, pp. 206–215. https://hal.archives-ouvertes.fr/hal-00920875

[7] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *HiPC, 2014*. 1–10. https://doi.org/10.1109/HiPC.2014.7116904

[8] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory. (2007).

[9] Google. 2015. Googl Heap Profiler, Google Perf-tool. (2015).

[10] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*. 125–138.

[11] Kim Hazelwood, Greg Lueck, and Robert Cohn. 2009. Scalable Support for Multithreaded Applications on Dynamic Binary Instrumentation Systems *(ISMM '09)*. ACM, 20–29. https://doi.org/10.1145/1542431.1542435

[12] Andreas Knüpfer and al. 2012. *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*. Chapter Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir, 79–91.

[13] Nicholas Nethercote and Julian Seward. 2007. How to Shadow Every Byte of Memory Used by a Program *(VEE '07)*. ACM, 65–74. https://doi.org/10.1145/1254810.1254820

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation *(PLDI '07)*. ACM, 89–100. https://doi.org/10.1145/1250734.1250746

[15] N. Nethercote, R. Walsh, and J. Fitzhardinge. 2006. "Building Workload Characterization Tools with Valgrind". In *Workload Characterization, 2006 IEEE International Symposium on*. 2–2. https://doi.org/10.1109/IISWC.2006.302723

[16] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and Precisely Locating Memory Leaks and Bloat *(PLDI '09)*. ACM, 397–407. https://doi.org/10.1145/1542476.1542521

[17] B. Perens. 1993. efence. (1993).

[18] Nathalie Rauschmayr. 2015. FOM-Tool: Find Obsolete Memory. (2015). https://gitlab.cern.ch/fom/FOM-tools

[19] L. Robertson. 2004. Anecdotes. *IEEE Annals of the History of Computing* 26, 4 (Oct 2004), 71–73. https://doi.org/10.1109/MAHC.2004.22

[20] Andreas Schäfer and Dietmar Fey. 2008. LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes. In *PVM/MPI*. 285–294.

[21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[22] Sameer Shende, Allen D. Malony, Alan Morris, and Peter H. Beckman. 2015. Performance and Memory Evaluation Using TAU. (2015).

[23] CERN LHCb team. 2015. (2015). https://gitlab.cern.ch/lhcb/Gaudi

[24] L Tuura, V Innocente, and G Eulisse. 2008. Analysing CMS software performance using IgProf, OProfile and callgrind. 119, 4 (2008), 042030. http://stacks.iop.org/1742-6596/119/i=4/a=042030

[25] Sébastien Valat, Marc Pérache, and William Jalby. 2013. Introducing Kernel-level Page Reuse for High Performance Computing *(MSPC '13)*. ACM, New York, NY, USA, Article 3, 9 pages. https://doi.org/10.1145/2492408.2492414

[26] Josef Weidendorfer. 2004. Performance Optimization: Simulation and Real Measurement *(aKademy)*.

[27] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588