Memory Allocation Tracing with VampirTrace

Matthias Jurenz, Ronny Brendel, Andreas Knüpfer,
Matthias Müller, and Wolfgang E. Nagel

ZIH, TU Dresden, Germany

Abstract. The paper presents methods for instrumentation and measurement of applications' memory allocation behavior over time. It provides some background about possible performance problems related to memory allocation as well as to memory allocator libraries. Then, different methods for data acquisition and representation are discussed. Finally, memory allocation tracing integrated in VampirTrace is demonstrated with a real-world HPC example application from aerodynamical simulation and optimization.

Keywords: Tracing, Performance Analysis, Memory Allocation.

1 Introduction

High Performance Computing (HPC) aims to achieve optimum performance on high-end platforms. Always, the achievable performance is limited by one or more resources, like available processors, floating point throughput or communication bandwidth. Memory is another important resource, in particular for data intensive applications [8].

This paper presents methods for instrumentation and measurement of program's memory allocation behavior. The integration into VampirTrace [9] provides additional information for trace-based performance analysis and visualization.

The rest of the first Section discusses the influence of memory allocation to performance, introduces memory allocators and references some related work. The following Sections 2 and 3 show various instrumentation approaches and ways of representing the result data. In Section 4 the approach is demonstrated with a real-world application example before the final Section 5 gives conclusions and an outlook on future work.

1.1 Impact of Memory Allocation on Application Performance

Memory consumption may cause notable performance effects on HPC applications, in particular for data intensive applications with very large memory requirements. There are three general categories of memory allocation related performance issues:

- memory requirements as such
- memory management overhead
- memory accesses

Firstly, excessive memory requirements by applications exceeding the available resources may lead to severe performance penalties. An oversized memory allocation

Y. Shi et al. (Eds.): ICCS 2007, Part II, LNCS 4488, pp. 839–846, 2007.

request might make an application fail or cause memory paging to secondary storage. The latter might happen unnoticed but brings severe performance penalties.

Secondly, memory management may cause unnecessary overhead - see also Section 1.2. This might be either due to frequent allocation/deallocation or due to memory placement which can cause so called memory fragmentation.

Thirdly, accesses to allocated memory may cause performance problems. In general, tracing is unsuitable to track single memory accesses, because of the disproportional overhead. This is covered by many tools already by means of hardware counters [4].

1.2 Memory Allocators

The memory allocator implements the software interface for applications to request memory. For example in C it contains the `malloc`, `realloc` and `free` functions and few more. Only the memory allocator is communicating with the operating system kernel to actually request additional memory pages in the virtual address space.

Usually, memory allocators handle small requests differently than medium or large requests. This is done for two reasons. Firstly, because the operating system can partition memory only in multiples of the virtual memory page size. And secondly, to reduce run-time overhead for small requests.

Reserving a whole page for very small requests would cause a lot of overhead. Thus, multiple small requests can be placed in the same page. This page can only be released again, if all memory requests in it have been freed (memory fragmentation). However, the memory allocator can re-use freed parts for following requests with matching sizes.

All allocators add a small amount of management data to the memory blocks delivered. Usually, this is one integer (or few) of machine's address size, see Section 2.3 below. Furthermore, memory allocators will pre-allocate a number of pages from the operating system for small requests. By this means, it avoids issuing an expensive system call for every single small request. This provides a notable performance optimization esp. in multi-threaded situations [1].

Memory allocators come as part of a platform's system library (libc) or as external libraries that can be linked to an application explicitly. Well known examples are the Lea allocator and the Hoard allocator library [1, 2].

1.3 Related Work

There is a large number of memory debugging tools available that check memory allocation/deallocation or even memory accesses. While such tools serve a different purpose than the presented work, they use similar techniques to intercept the allocation calls.

One typical and small example is the *ccmalloc* library for debugging memory management and accesses. It uses dynamic wrapper functions to intercept memory management function calls [3]. The *MemProf* tool allows memory usage profiling [14]. It provides summary information about memory allocation per function. *Valgrind* is a very powerful memory debugger and profiler [15]. It offers more sophisticated debugging techniques to detect bugs in memory allocation/deallocation as well as invalid accesses. Unlike the previous examples, Valgrind depends on simulation of target applications.

Even though, this is highly optimized, it causes notable slowdown of target applications which is well acceptable for debugging but not for parallel performance analysis.

Besides debuggin tools, there are performance analysis tools focusing on memory allocation. The TAU tool set offers special operation modes for profiling and tracing of memory allocation [13]. It employs preprocessor instrumentation for memory management calls and takes advantage of platform specific interfaces (Cray XT3). Also, it introduces the *memory headroom* metric for remaining heap memory, see Section 2.3.

Other existing performance measurement and analysis tools focus on memory access behavior. This is done either with summary information provided by PAPI counters [4] or with even more detailed information which are achievable by simulation only [6].

2 Instrumentation of Allocation/De-Allocation Operations

Instrumentation is the process of inserting measurement code into a target application. There are various general instrumentation techniques. They can be applied at source code level, at compilation or linking time or at binary level. The different approaches vary in terms of platform dependence, in terms of measurement overhead and even in terms of expressiveness. Below, some approaches are discussed with special focus on analysis of memory allocation.

2.1 The `proc` File System

Firstly, there are interfaces to query general process information. One of the most commonly known is the `proc` file system. It provides system wide information as well as process specific data. In `/proc/meminfo` the total available memory of the system is given among other details. This might not always be the amount of memory actually available to a process because of user limits, machine partitioning or other means.

In `/proc/<PID>/statm` there is information about the memory usage of the process with the given `PID`. It provides current memory size of heap and stack in multiples of the page size (usually 4 KB). This interface is widely supported but is rather slow in terms of query speed. For our experiments we use it optionally to determine total memory consumption.

2.2 The `mallinfo` Interface

The `mallinfo` interface, which provides detailed allocation statistics, is provided by many allocators. Although it returns all desired information, is platform independent and has fast query speed and yet it is unusable for our tool. Unfortunately, it uses 32 bit integers by design, thus being incapable of reporting about memory intensive applications in 64 bit address space.

2.3 Autonomous Recording

An autonomous approach can be implemented by intercepting all memory allocation operations via wrapper functions[1]. Wrappers evaluate the function's arguments, e.g.

[1] Wrapper functions replace calls to a target but call it again itself. Besides the original purpose of the function, additional tasks like logging, measurement, checking, etc. can be issued.

memory size in `malloc()`, which are essential for recording memory consumption. However, for some calls the corresponding memory sizes are unknown, e.g. for `free()`.

It is necessary to infer the size of deallocated memory area at this point in order to correctly update the record. Explicitly storing this information would create memory overhead. Fortunately, memory managers do provide it. Usually, it is stored as an integer at the position right in front of the very memory area. It can be accessed either by the `malloc_usable_size` function or directly. Note that this deals with the memory allocated, which is greater than or equal to the amount requested. For consistence out approach always reports the amount of memory actually used, including memory management overhead. This is greater or equal to user application requests.

Internally, the amount of memory currently allocated is stored in thread-private variables within the measurement infrastructure. Multiple threads in the same process will record their memory consumption separately. This avoids unnecessary thread synchronization. Below, there is a short discussion of different ways to intercept all memory management calls like `malloc`, `realloc`, `free`, etc.

Pre-Processor Instrumentation. Compiler pre-processors can be used to replace all function calls in the source code with given names by alternative (wrapper) functions. Those need to be provided by the the trace library.

With this approach it is possible to miss certain allocation operations hidden in library calls with inaccessible sources. This can cause falsified results. For example, memory allocated within third party libraries might be freed by the user application directly. Therefore, this approach is not advisable without special precautions.

Malloc Hooks. The GNU glibc implementation provides a special hook mechanism that allows intercepting *all* calls to allocation and free functions. This is independent from compilation or source code access but relies on the underlying system library. Unlike the previous method, this is suitable for Fortran as well. It is very useful to guaranty a balanced recording, i.e. all allocation, re-allocation and free operations are captured. Similar mechanisms are provided by other libc implementations, e.g. on SGI IRIX.

This approach requires to change internal function pointers in a *non-thread-safe* way! It requires explicit locking if used in a multi-threaded environment. Nevertheless, this is the default way for MPI-only programs in our implementation.

Library Pre-Load Instrumentation. An alternative technique to intercept all allocation functions uses pre-loading of shared libraries. Within the instrumentation library there are alternative symbols for `malloc`, `realloc`, `free`, etc. which are preferred over all later symbols with the same name. The original memory management functions can be accessed explicitly by means of the dynamic loader library. This approach requires support for dynamic linking, which is very common but not available on all platforms (e.g. IBM's BlueGene/L). Unlike the previous approach, it can be implemented in a thread-safe manner without locking.

Exhausting Allocation. There is another remarkable strategy to detect the available amount of heap memory, the so called *memory headroom*. It counts the free heap memory and infers about the consumed memory (implying the total amount of memory

is constant) [13]. Free heap memory is estimated by exhaustingly allocating as much memory as possible (with a $O(\log n)$ strategy). Presumably, this approach has a big impact on run-time overhead as well as the memory manager's behavior. Furthermore, it will fail with optimistic memory managers like the default Linux one.

2.4 Measurement Overhead

Overhead from instrumentation can occur in terms of additional memory consumption and in terms of run-time. There is almost no memory overhead from allocation tracing with the exception of one counter variable (64 bit integer) per process.

The run-time overhead of malloc hooks is not detectable within the given accuracy of measurement of $\approx 5\,\%$. It was tested on single-processor AMD Athlon64 machine with Linux OS and with a pathological worst case test. It issues $20,000,000$ allocation requests of differing sizes without accessing the memory or any real computation. See Figure 1 for run-time results including wall clock time, system time and user time.

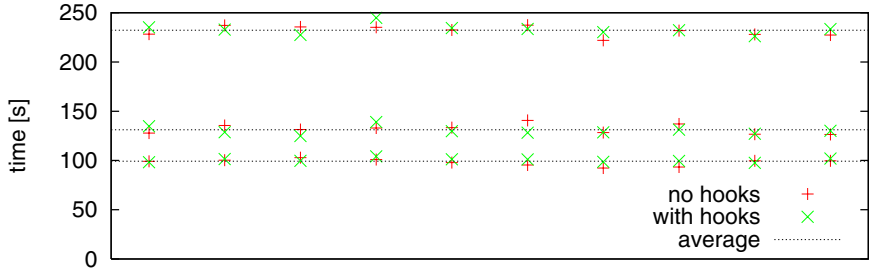

Fig. 1. Run time overhead for instrumentation. The figure compares time without and with instrumentation for ten test runs with a pathological program. It shows total wall clock time (top), system time (middle) and user time (bottom) as well as the respective average values.

The test covered instrumentation only, no trace records were generated neither in memory buffers nor in actual trace files. The total tracing overhead may slightly increase due to the additional records which makes the records buffer flush slightly more often.

3 Representation of Result Data

There is a common conception how various trace file formats store trace information. All employ so called *trace records* as atomic pieces of information. Representation (syntax) and expressiveness (semantics) vary between the existing formats, though [11]. There are two obvious options, how to represent memory allocation information. Firstly, to extend (a) trace format(s) by a novel record type with specified semantics. Secondly, to re-use an existing record type which is suitable to transport the information.

3.1 Novel Record Types

One or more new record types could be introduced specifically tailored towards memory allocation information. It might be special flavors of enter and leave record types enhanced by allocation information or special record types carrying only allocation information, that are to be used in addition to existing enter and leave records. There may be several sub-types for allocation, re-allocation and de-allocation or a common one.

Both ways combine the same major advantage and disadvantage. On one hand, they allow to express more semantics, i.e. the presence of memory allocation information can reliably be determined by a specific record type. On the other hand, they would require any trace analysis tools to adapt explicitly to the new record types. We regard this as a major obstacle for general acceptance by performance analysis tool developers.

3.2 Generic Performance Counters

The memory allocation over time can be mapped to ordinary performance counter records [4, 9]. Performance counter records are designed to represent scalar attributes over discrete time, which is very suitable for memory allocation information. Usually, for every enter and leave event there needs to be a samples with matching time stamp for every active counter. This results in two samples per function call per counter. Only so, the counter values can be associated to function calls.

This is unfavorable for memory allocation information because it changes rather infrequently. Instead, the memory allocation counter is only updated on actual changes. Every update of this counter specifies a constant value for memory consumption from the current time until the next sample[2]. Thus, the memory consumption is represented as a step function (piecewise constant) with very low overhead in terms of extra records.

4 Application Example

The example that memory allocation tracing was tested with originates from an aerodynamics application. The sequential application consists of two components for simulation and for optimization of the simulation results. The first part is the flow solver TAUij from the German Aerospace Center (DLR), which is a quasi 2D version of TAUijk [2]. It solves the quasi 2D Euler equations around airfoils and computes the aerodynamic coefficients for drag, lift and pitching moment.

The second part uses the ADOL-C software package for Automatic Differentiation (AD). ADOL-C attaches to TAUij by overloading variable types and operators and provides mechanisms to evaluate various kinds of derivations of the original program's computation. Here, it is used to compute the gradient of the drag coefficient with respect to the input variables controlling the wing geometry. The gradient is then used for optimizing the wing shape, which involves iterated simulation and gradient evaluation. See [5, 7, 12] for more details and background information.

In the example the so called *reverse mode* of AD is applied to compute the gradient with respect to a large number of input variables. Basically, this requires to store *all* intermediate results of the computation in a so called *memory tape*. This is to be traversed

[2] This behavior can be announced by setting a corresponding flag in the counter definition [10].

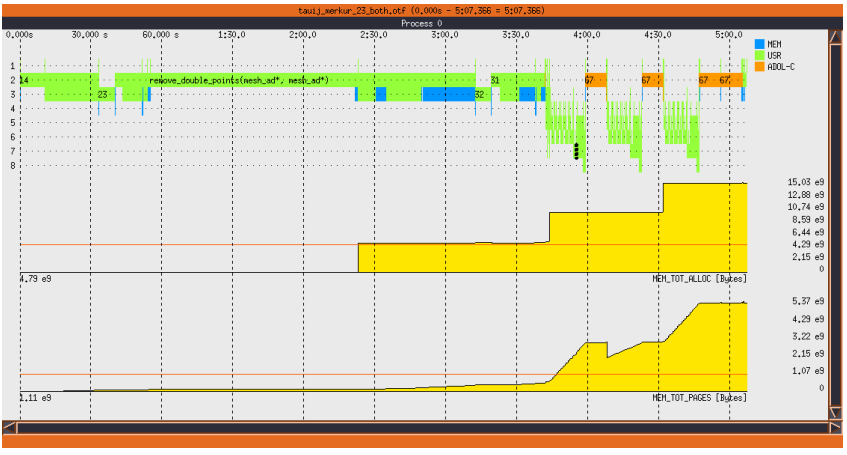

Fig. 2. Vampir process timeline with counters `MEM_TOT_ALLOC` and `MEM_TOT_PAGES`

in reverse order in the stage of gradient computation. There are some optimizations for fixpoint iteration algorithms, but still it causes excessive memory requirements [7].

This is shown in the Vampir screenshot in Figure 2. The test-run consumes ≈ 15 GB of memory (`MEM_TOT_ALLOC`). The exact amount is neither constant nor easily predictable. The analysis revealed that the actual usage is only ≈ 5.4 GB (`MEM_TOT_PAGES`).

The experiments involved additional PAPI counters that showed that the memory intensive sections of the application suffer from high miss rates of level three cache due to the linear traversal of the memory tape.

5 Conclusion and Outlook

The paper discussed various methods for memory allocation tracing for HPC applications. They have been implemented in the VampirTrace measurement system [9] and used with a real-world example application. It was able to provide some insight in the program's run-time behavior, that was not obviously visible without this approach.

Future work will focus on alternative ways to query kernel statistics about memory pages assigned to processes. We would like to decrease the run-time overhead. Special kernel modules might be a solution. This is currently attempted for I/O tracing.

We will apply memory allocation tracing to more HPC programs in the course of performance analysis to learn more about typical memory management behavior. The convenient integration into VampirTrace allows this by adding a mere run-time option.

Acknowledgments

We'd like to thank Carsten Moldenhauer and Andrea Walther from IWR, TU Dresden as well as Nicolas R. Gauger and Ralf Heinrich from the Institute of Aerodynamics and Flow Technology at DLR Braunschweig for their support with the example application.

References

[1] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. of ASPLOS-IX*, Cambridge, MA, 2000.

[2] E.D. Berger, B.G. Zorn, and K.S. McKinley. Reconsidering custom memory allocation. In *Proc. of OOPSLA'02*, New York, NY, USA, 2002. ACM Press.

[3] Armin Biere. *ccmalloc*. ETH Zurich, 2003. `http://www.inf.ethz.ch/personal/projects/ccmalloc/`.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[5] N. Gauger, A. Walther, C. Moldenhauer, and M. Widhalm. Automatic differentiation of an entire design chain with applications. In *Jahresbericht der Arbeitsgemeinschaft Strömungen mit Ablösung STAB*. 2006.

[6] M. Gerndt and T. Li. Automated Analysis of Memory Access Behavior. In *Proceedings of HIPS-HPGC 2005 and IPDPS 2005*, Denver, Colorado, USA, Apr 2005.

[7] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22:131–167, 1996.

[8] G. Juckeland, M. S. Müller, W. E. Nagel, and St. Pflüger. Accessing Data on SGI Altix: An Experience with Reality. In *In Proc. of WMPI-2006*, Austin, TX, USA, Feb 2006.

[9] Matthias Jurenz. *VampirTrace Software and Documentation*. ZIH, TU Dresden, Nov 2006. `http://www.tu-dresden.de/zih/vampirtrace/`.

[10] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Proc. of ICCS 2006: 6'th Intl. Conference on Computational Science*, Springer LNCS 3992, pages 526 – 533, Reading, UK, May 2006.

[11] Bernd Mohr. Standardization of event traces considered harmful: or is an implementation of object-independent event trace monitoring and analysis systems possible? *Journal: Environments and tools for parallel scientific computing*, pages 103–124, 1993.

[12] S. Schlenkrich, A. Walther, N.R. Gauger, and R. Heinrich. Differentiating Fixed Point Iterations with ADOL-C: Gradient Calculation for Fluid Dynamics. In *Proc. of HPSC 2006*.

[13] S. Shende, A. D. Malony, A. Morris, and P. Beckman. Performance and memory evaluation using tau. In *Proc. for Cray User's Group Conference (CUG 2006)*, 2006.

[14] Owen Taylor. *MemProf*. `http://www.gnome.org/projects/memprof/`.

[15] Valgrind.org. *Valgrind*, 2006. `http://valgrind.org/info/about.html`.