Examensarbete

Memory Profiling Techniques

av

Andrei Faur

LIU-IDA/LITH-EX-A--12/021--SE

2012-06-20
Examensarbete

Memory Profiling Techniques

av

Andrei Faur

LIU-IDA/LITH-EX-A--12/021--SE

2012-06-20

Handledare: Adrian Lifa / Fredrik Söderquist
Examinator: Petru Eleș
To my parents, whose constant love and support have been invaluable throughout the years:

Thank you, you are my lighthouse in this storm.
Abstract

Memory profiling is an important technique which aids program optimization and can even help tracking down bugs. The main problem with the current memory profiling techniques and tools is that they slow down the target software considerably therefore making them inadequate for mainline integration. Ideally, the user would be able to monitor memory consumption without having to worry about the rest of the software being affected in any way. This thesis provides a comparison of existing techniques and tools along with the description of a memory profiler implementation which tries to provide a balance between the information it is able to retrieve and the influence it has on the target software.
Contents

1 Introduction 1

2 Memory Management Concepts 3
  2.1 Virtual Memory ........................................ 4
  2.2 Memory Layout ......................................... 5
  2.3 The Heap ................................................. 6
  2.4 The Stack ................................................. 8

3 Memory Analysis Methods 10
  3.1 Profiling Methods ...................................... 11
    3.1.1 Code Instrumentation ............................... 12
    3.1.2 Statistical Profiling ............................... 14
    3.1.3 Performance Counters .............................. 15
    3.1.4 Hardware-assisted Profiling ....................... 16
    3.1.5 Event-based Profiling .............................. 17
  3.2 Heap Profiling .......................................... 17
    3.2.1 Allocation Size Profiling ......................... 18
    3.2.2 Allocation Point Profiling ....................... 21

4 Results and Solution 27
  4.1 Test Results ............................................ 28
    4.1.1 Test Infrastructure Description .................. 28
    4.1.2 Allocation Size Overhead Results ................. 30
    4.1.3 Allocation Point Overhead Results ............... 31
    4.1.4 Other Issues ........................................ 34
  4.2 Solution Description .................................... 35

5 Conclusion 39

Bibliography 41
List of Figures

2.1 Two processes mapping memory in diverse ways . . . . . . . 4
2.2 Typical virtual memory layout . . . . . . . . . . . . . . . 5
2.3 Typical stack layout . . . . . . . . . . . . . . . . . . . . . 9

4.1 Allocation size time overhead compared to the basic scenario
   when no logging is done . . . . . . . . . . . . . . . . . . . . 31
4.2 Allocation size time overhead compared to the basic scenario
   when logging is done . . . . . . . . . . . . . . . . . . . . . 32
4.3 Allocation point time overhead compared to the basic scenario 33
Chapter 1

Introduction

In large and complex programs such as web browsers, word processors, enterprise software and most modern popular tools, it is not trivial to map memory usage to responsible subsystems. The high number of such subsystems (e.g. in a web browser: page loading, ECMAScript, DOM, HTML Layout and Rendering), their interaction and their different memory requirements make such an analysis difficult. Several memory allocators might be used, depending on each subsystem’s requirements. For example, one part of the software might need quick access to a large number of small, fixed-size chunks of memory. An allocator that caters to that need could improve performance and therefore be used. Different platforms have different ways of organizing memory which cannot be ignored if detailed memory consumption information is required. This diversity, coupled with the sheer size of modern software’s codebase (millions of lines of code) excludes any trivial solutions.

Having memory usage information available can lead to improvements in subsystems that exhibit constant high memory consumption and could potentially lead to bug discovery in subsystems which show temporary memory usage spikes or unusual memory consumption patterns. In addition, having detailed memory tracking logs from different instances of a piece of software running on diverse platforms could lead to platform specific optimizations.

Monitoring the way memory management is performed by a piece of software involves, among others, heap profiling, real-time fragmentation visualization, allocation and deallocation performance, overallocation detection and memory leak detection; all of this information should presented for each of the software’s subsystems as well as at a global level. The user of this information should have access to detailed information related to the allocation site which exhibited unusual behaviour, such as: module membership, stack traces and exact size of memory allocated at the site. Each of the previously mentioned issues present problems on their own:
• Heap profiling requires detailed information on how allocations are performed: by whom, exact size, possible stack traces.

• Measuring allocation and deallocation performance requires keeping track of time.

• Fragmentation visualization must keep track of the exact layout of the heap and have in-depth knowledge of how allocators use it.

• Overallocation and memory leak detection involve checking every memory access.

The main problem is that any sort of measurement automatically reduces the software speed either through added code or by running it in a special environment. The challenge is to thus create or use a tool that is as noninvasive as possible and has minimal impact on speed so that it can be deployed in default installations. Even though all of the above approaches (i.e. heap profiling, fragmentation, etc.) have a common goal, which is to describe memory from a certain point of view, they do exhibit different requirements: if we are not interested in allocation performance we do not need to worry about time, therefore removing the overhead of timer management; simple heap profiling needs no knowledge of the heap layout and so on. Having a tool which performs all of these tasks, and does so in a way that does not affect performance at all is unlikely, given the fact that there exist separate current state-of-the-art tools and methods for each of the above and each of them has a negative impact on performance.

This thesis concerns itself primarily with heap profiling with the added constraint of good real-time performance. The goal is to obtain a solution which gives us detailed information about heap usage and possibly the chain of events which lead to a given state of the heap. We are interested in determining the following:

• How heavily classes/modules are using the heap (how much memory they allocate).

• The exact way in which they are using it (which methods are responsible for allocations).

• How they interacted with each other (what are the call chains that lead to the allocations).
Chapter 2

Memory Management Concepts

Throughout the lifetime of a process its memory requirements change. Whether the process has to create more objects or allocate arrays or even temporary variables, it has to have a way of requesting more memory and a way to release that memory when it is no longer needed. Since our purpose is to actively monitor the exact memory consumption of a process\(^1\), the underlying mechanisms of memory allocation and deallocation are of direct interest. This chapter explains these mechanisms and explains how each of them is relevant to our original goals of finding out how much memory a process consumes and how different parts of that process interact with each other to reach that specific memory consumption state. The terms described in this chapter will be used throughout the rest of the thesis and they constitute the framework of our problem.

\(^1\)For simplicity, let us assume that the piece of software we are interested in monitoring runs only in one process.
2.1 Virtual Memory

Virtual memory is a mechanism used by modern operating systems in order to give processes the illusion that there exists only one type of memory in the system which exhibits the behaviour of a directly addressable read/write memory. In addition, most operating systems run processes in separate address spaces providing the impression that processes have exclusive access to the virtual memory\(^2\). This is accomplished by the operating system by avoiding the direct use of physical addresses and instead making processes use logical addresses which then get translated by the operating system and the memory management unit into physical addresses. Figure 2.1 shows a 32-bit system with two processes and their address spaces and the way they are mapped to physical memory and other devices.

Note that the virtual memory mechanism allows different process to share memory between each other by mapping the exact area of physical memory into possibly different areas of virtual memory and also map other devices into their address space, including files from the disk. The same virtual address in different processes can be mapped to different places, enforcing the idea of address space separation. The exact mechanisms which make virtual memory work and concepts such as translation lookaside buffer, paging, multi-level page tables, page replacement algorithms have been described in great detail in OS literature such as [1], [2], [3] and many more. Since these

\(^2\)There exist operating systems which use a single global address space, such as OS/VS1 and IBM i, but they still include mechanisms by which processes are stopped from accessing each other’s addresses
exact details do not have an impact on our analysis, the reader is referred to the references for more in-depth knowledge.

Monitoring a process’ memory thus becomes a problem of monitoring the way its virtual memory is mapped. This leads to the question of determining which area of a process’ virtual memory we are interested in monitoring. Do we monitor all of it or just specific parts? In order to answer that question we first have to understand exactly how a process’ virtual memory is organized.

2.2 Memory Layout

In order for a program to become a process it has to be loaded into memory\(^3\) by a part of the operating system called the loader. The question is then how is the process’ virtual address space organised? Several formats have existed over the years, such as Unix’s a.out, MS-DOS’s COM and the more recent ELF format. While these might differ drastically in terms of the object code representation, ultimately their goal is to produce a memory layout similar to Figure 2.2.

![Typical virtual memory layout](image)

**Figure 2.2: Typical virtual memory layout**

The segments represented are:

- text segment - which contains the actual code;

\(^3\)In systems with virtual memory no bytes of the program are actually copied into main memory but rather a part of the newly created process’ address space is marked as containing the code. Only when the code will be executed will it be brought into main memory.
• initialized data segment - global variables which are initialized by the programmer;
• uninitialized data segment - variables in this segment are initialized to 0 or NULL before the program begins to execute;
• the heap - used for allocating more memory during runtime, described in section 2.3;
• the stack - used for function calls, as described in section 2.4.

The text segment and static data segments (initialized and uninitialized) usually do not change in size during the lifetime of a process so they are of little interest; their size can be determined from compile time and can be reported easily. The stack and the heap, which usually grow towards each other, are constantly changing but their purpose is different. They will both help us in reaching our goal and, as it will be seen in the next subchapters, ultimately we will be interested in monitoring the heap, using the stack just as a source of additional data.

2.3 The Heap

The heap is where all dynamic allocations\(^4\) done during the lifetime of a process are stored. Operating systems offer system calls which expand the heap, thus providing access to more memory. For example, Linux offers the \texttt{brk} and \texttt{sbrk} system calls which change the location of the end of the process’ data segment, while Windows has the \texttt{*Alloc} system calls. It is, however, rare for high level applications to call these routines directly; instead, they use external libraries or libraries provided by the language they are written in. For example, the classical ways of allocating memory in UNIX systems makes use of the following standard C library calls:

• \texttt{malloc/calloc} - allocates a number of bytes from the heap and returns a pointer to the beginning of the block; calloc initializes this region to zero;
• \texttt{realloc} - given a pointer to a previously allocated block, expands that block by a given size; it is not guaranteed that the resulting block lies in the exact same place on the heap since there might not be enough contiguous space after the block;

\(^4\)Note that there exist system calls such as \texttt{alloca} which allow dynamic allocation of space on the stack. These are rarely used and as such they won’t be taken into consideration
• **free** - given a pointer to a previously allocated block, release that block and mark the memory as free.

Programs written in C++, even though able to call the above routines, make use of the *new* and *delete* operators. These operators however, in the standard C++ library, ultimately translate into calls to the above.

An additional system call available in Linux for mapping memory is `mmap` which is more flexible than `brk`. It allows mapping of any region of the virtual memory not only to RAM, but to files too. Given how everything in Linux is modeled as a file, including devices, `mmap` can basically map virtual memory to any device’s internal memory as long as the latter allows it. The `munmap` system call does the reverse process of unmapping virtual memory.

In order to monitor a process’ memory consumption we have to either hook the above calls or provide wrappers around them. By doing either of these we can answer our original question of how much memory an allocation site is requesting. By allocation site we refer to a point in a program where one of the above routines/operators is invoked.

Another interesting point worth mentioning is the problem of which area of virtual memory will be selected for mapping when one of the above routines is called. It is the job of the memory allocator to select the locations in such a way as to minimize fragmentation, maximize cache locality and provide fast allocation and deallocation speed at the same time[4]. These goals sometimes clash and trade-offs have to be made. Techniques such as reference counting, pooling and garbage collection are sometimes used in conjunction with allocators in order to lessen the burden of memory management from the programmer[5]. All of these have to be taken into consideration in order to do correct memory profiling. Communication with the garbage collector, for example, might be the only way to detect when memory gets deallocated since memory is no longer explicitly released. Another example could be allocators which preallocate memory in advance so that subsequent memory allocation requests are faster. In this case we have to ask ourselves if we are interested in every mapped byte from virtual memory or just those bytes which can be potentially accessed?

Memory allocators use different techniques for selecting which memory region to map when an allocation is requested, such as:

• **Memory pooling** - one or several chunks of the address space are requested by the allocator in advance. Subsequent allocation requests will be given chunks from these pools, thus avoiding unnecessary system calls. Different pools might have different purposes such as one targeted towards small allocations while the others are structured in
such a way as to minimize fragmentation. The advantage with pools is that they can usually be cleared using a single call and some implementations even allow their creation and destruction at will. This takes some of the burden of memory management from the programmer. Keeping track of every allocation is no longer necessary since a pool can be destroyed by just a simple call and all previous allocations made from that pool are instantly gone.

- **The buddy system** - this technique divides the address space into blocks which have the size multiple of powers of two. The initial size of these blocks and their minimum size is usually platform dependent and also chosen empirically, based on most common allocation sizes. When an allocation is made, of size smaller than the smallest block available, one of these blocks is chosen and then divided in two. The process is repeated until the division would lead to the creation of blocks of smaller size than the allocation requested. One of the blocks thus created is then given to the allocation. The name of this technique comes from what happens when a block is freed. All the previous divisions created "buddy blocks" of equal sizes. When one of these blocks is freed then its buddy is checked to see if it is free. If it is, then the two are joined into a larger block and the process repeats.

One thing that most allocators have in common is that they need a way of keeping track of which parts of the memory have been allocated. For example, in the buddy system, either linked lists or trees can be used to store blocks which have the same size. When memory pooling is used, there is both a need of keeping track of all the pools available and keeping track of how each pool is organized. A simple allocator might just use two lists: one keeping track of all the available chunks of memory and their sizes and one for those which are allocated. One thing which is clear is that all this bookkeeping does incur overhead both in time and in terms of used memory. Moreover, for correct memory profiling, close communication with these allocators might be required in order to see how memory is allocated.

### 2.4 The Stack

The stack is used by processes to keep track of the call chain. Each time a function is called a new stack frame is created and pushed on the stack. Upon return from that function, the stack frame is popped. A stack frame’s exact structure is dependant on the platform on which the code is running, but in all cases it, at least, contains the following:
- the arguments passed to the function;
- the return address back to the caller’s code;
- space for the function’s local variables;
- space for the function’s temporary variables in case they can not be stored in registers.

Let us assume we have a process whose current call chain is main()-f1()-f2(). The process’ stack and its evolution are illustrated in Figure 2.3.

The information contained in the stack at any point in time allows us to trace the call chain from the current execution point all the way to a program’s entry point. Suppose that f2() in Figure 2.3 did an allocation in which we were interested. By using the stack, we are able to go back from an allocation site through as many callers as we want. This answers one of our original questions: who is responsible for allocating memory?

Figure 2.3: Typical stack layout
Chapter 3

Memory Analysis Methods

In this chapter we will focus on the description of methods used in obtaining dynamic memory allocation information. We will start by presenting approaches used by profilers in general and see how these are applicable to memory profiling in particular. Given the fact that we are concerned with both the size and the locations of memory allocations and since these two involve different approaches, we will present different methods of obtaining relevant information separately for these two. Finally, a test program and the platform on which it was run will be described, upon which all these methods will be tested. This will provide a good performance reference in order to determine the overhead memory profiling techniques induce.
3.1 Profiling Methods

Profiling a computer program means analyzing its behaviour during runtime in order to obtain information that can lead to its optimization. The sought information can vary from memory usage to identifying the most commonly taken call paths, all the way down to cache misses and the use of particular processor instructions. Any interaction between the program and the OS and, consequently, the hardware, can potentially be analyzed and, based on the results, improved if possible.

We can make a first classification of profilers based upon their intrusiveness. Thus, we have intrusive profilers which modify the program’s instructions in some way, in order to insert code which helps with the analysis. These profilers can be classified further into those that need a program’s source code to perform the analysis and those that can work on a program’s binary form. Finally, we have non-intrusive profilers which require no modification of the original program.

Another classification can be made based on the technique profilers use to collect information. We can thus have:

- **Code instrumentation** profilers which add or modify instructions in the original program to collect the required information.

- **Statistical** profilers which work by periodically sampling the program and then extrapolate conclusions statistically.

- **Performance counters** are special registers in modern processors which keep track of specific events such as cache misses. These can be used to observe behavioral patterns in programs.

- **Hardware assisted** profilers use dedicated hardware to collect and analyze information about running programs.

- **Event based** profilers use predetermined hooks provided by the underlying software and hardware platforms to collect data on the running program.

This classification is not precise as many profilers use, for example, both performance counters and code instrumentation to create more detailed profiles. In the following sections we will describe these techniques in more detail and see how and if they relate to our goal of low overhead memory allocation profiling.
3.1.1 Code Instrumentation

Code instrumentation is the process of altering a program (by adding code or modifying existing code) in order to collect performance statistics. There are several ways in which this can be done.

Source level instrumentation

This involves modifying a program’s actual source code before or during compilation. Having direct access to a program’s source code has the big advantage of being able to monitor application-specific statistics. We can further classify this into the following:

- **manual** instrumentation can be done by the programmer by simply adding instructions which monitor different statistics at points of his choosing. The advantage of this approach is that it can be very fast since it requires no external tool to be run alongside the program and the inserted code can be tailored to the application’s specifics so it can take advantage of things such as garbage collection runs. The disadvantage is that it requires deep knowledge of an application’s source code in order to identify the points where instrumenting code should be added. Thus, it is heavily intrusive but has the advantage of potentially having very low overhead.

- **tool assisted source level** instrumentation involves using external tools to insert instrumentation code in the program’s source code. For example, through a specific language the tool could be guided to monitor the number of times a specific function has been called during a run of the program. Thus, there is a shift from the programmer actually modifying the program’s source code to instructing a tool in what way it should modify the program so that the desired statistics are collected. The advantage of this approach over manual instrumentation is that no detailed knowledge of the source code is required but with this flexibility is lost. For example, monitoring complicated internal data structures might involve complex tool scripts or not even be possible at all. Another approach is to have a tool that analyzes the source code in order to indicate the best points for instrumentation code to be added. This approach has been described by Larus and Ball[6].

- **compiler assisted** instrumentation can be viewed as another form of tool-assisted instrumentation where the tool is the compiler itself. An example of this is GCC, which has the option of adding code to a
program thus allowing the program to output profiling information which can be analyzed offline by a call graph profiler called gprof\[7\]. The information provided by this tool is related to the time spent in each of the program’s functions and the way the functions interact with each other.

**Binary level instrumentation**

This involves modifying a program’s source code in binary form offline or during runtime. The main advantage of this approach is that it does not require access to the source code so programs that do not have their source publicly available can be analyzed too. The downside is that the complexity of these tools is large since the binaries have to be carefully analyzed before they are modified. Because of this and the fact that it is very difficult to draw conclusions about an application from its executable, application-specific monitoring (such as the data structure example above, or, another example, determining how much memory a tab consumes inside a web browser) is very difficult using this approach. It can be further categorized as follows:

- **Binary alteration** means modifying a program’s binary before it is run. ATOM\[8\] is a tool which allows instrumentation code to be added to applications using only link-time information. To be more exact it is a tool for building profiling tools. It works by providing a framework for the definition of instrumentation routines and for merging these routines with the program to create an instrumented executable. The LOPI framework\[9\] implements a similar solution.

- **Binary code injection** tools add code to a program while it is running. Dyninst\[10\], for example, uses a concept known as code trampolines to perform this task. The idea is that simply replacing code in the original binary with a jump instruction to the routine that performs the profiling is not possible because of the possibility of overwriting in-use registers. Thus, the jump instruction points instead to a piece of code known as a trampoline. This piece of code has the responsibility of saving the context from the jump point and restoring it after the profiling function has been executed. Several implementation schemes of this are possible such as the use of a separate mini-trampoline whose task is to execute the original replaced instruction. No matter the implementation, however, the idea of preserving the context and thus the correctness of the program remains. A similar technique, called springboards, has been attempted for use in kernel profiling\[11\].
unified solution for both userspace and kernelspace code profiling has been implemented by the DTrace tool[12].

- Runtime translation is a method which involves converting a program’s instructions into another representation which is more suitable for profiling. Valgrind[13] and PIN[14] are tools which implement this technique. Valgrind, for example, uses a method through which every register and memory value is shadowed and thus can be monitored. This allows for very powerful memory leak detectors to be implemented. The downside to the code conversion is that it incurs a significant time penalty, thus, even a tool that does nothing in Valgrind (called nullgrind) slows down the program significantly enough that it cannot be used for live analysis [15].

3.1.2 Statistical Profiling

Another technique used in profiling is to periodically sample the statistics we are interested in. For example, every X miliseconds, the program can be stopped and its stack trace can be inspected. By doing this multiple times, we can deduce how much time the program spends in every routine. Of course, the precision of this approach depends on the frequency of the sampling. The downside is that the overhead increases with frequency.

There exist different approaches to how the sampling can be done:

- Period sampling is the simplest method, where the period is chosen randomly and then modified empirically until a good balance between overhead and results has been reached. This approach has been used by Whaley[16] to implement a profiler for Java virtual machines which focuses on an efficient way of organizing and storing stack trace information.

- Bursty tracing uses two variables: one which specifies the sampling rate and another to specify how long the sampling should last. It is used in conjunction with instrumented code which is only executed when sampling is enabled. Adaptive versions of this technique have multiple sampling rates and durations for different code areas in order to selectively control the analysis frequency of those areas which are considered to be more important. Chilimbi and Hauswirth[17] have used this approach in order to implement memory leak detection by using a tree-based heap model which stores information about access frequency of objects on the heap. This access frequency is updated through sampling-enabled instrumentation code. Objects which have
not been accessed for a long time (either because they truly have not been accessed or the sampling missed their accesses) are reported. The idea of using two versions of the code and switching between them based on a sampling rate was originally presented by Arnold and Ryder[18].

- **Stride based sampling** uses three parameters: one for the sampling rate, one to specify a count-down mechanism for sampling every n-th method call (the stride) and another one to give the length of the profiling window. The sampling rate is usually determined by a timer, eliminating the need of maintaining a counter. This approach has been used by Arnold and Grove[19] to implement call graph profiling in virtual machines.

Tools which rely on statistical profiling for some of the information they provide are AMD CodeAnalyst, Intel VTune and gprof, which we mentioned earlier when discussing code instrumentation. Gprof actually uses sampling to determine the time spent in certain functions while it uses counting based instrumentation methods to keep track of how often a certain function has been called.

Using sampling in order to determine the exact memory consumption of a program would mean sampling a number of allocations for a certain period of time and then deducing the total number of bytes used from those allocations. There is the possibility of the profiling window completely missing important allocations so precise calculations are not possible. It would be possible to determine the average number of allocated bytes per unit of time and then draw conclusions from that but, again, we are interested in byte-level precision. However, as later subchapters show, there is a role that sampling based profiling plays in our technique and that is to determine how often we should trigger computations which aid in determining a program’s running size.

### 3.1.3 Performance Counters

Performance counters are special registers found on modern platforms which keep track of CPU cycles, completed instructions, instruction cache misses, data cache misses, TLB misses and many more. They either count events or cycles. Cache counters usually do both, the first one for keeping the number of cache misses and the second for keeping the total number of cycles lost due to these misses. Some architectures provide counters which are configurable. These counters are not tied to monitoring a specific event but they can be configured to monitoring any event from a pre-determined list.
Itzkowitz and Wylie[20] describe the difficulties of using performance counters, including a solution for handling their overflow. They provide an implementation of a data collector and analyzer which ties performance counters’ values with discrete instructions from a given program.

London and Moore have proposed a unified framework for cross-platform hardware performance counters accessibility[21]. This framework aims to abstract away the low-level details of accessing the counters and to provide their values in a uniform way across different platforms.

While these counters by themselves provide very little information directly related to memory allocations, they can be used as data which drives other tools to implement memory optimizations. For example, Tikir and Hollingsworth[22] used such counters to profile the memory access behaviour of an application and then based on this profile, move the most frequently accessed memory pages into caches closer to the processor. However, while it is possible to use performance counters to determine an application’s memory profile, information directly related to allocation sizes and points is usually found at layers above the hardware level. Thus, their use in precisely determining allocation information is limited.

OProfile is a tool which allows fine-grained hardware counter monitoring on Linux. It combines access to a wide array of counters on different platforms with statistical profiling to allow from instruction-level up to function-level profiling.

### 3.1.4 Hardware-assisted Profiling

One step forward from performance counters is to have complex dedicated hardware components which aid the profiling process. Rather than being just simple counters, these hardware components can range from simple auxiliary microprocessors to completely using existing processors from multi-core architectures for profiling purposes.

Different hardware approaches have been implemented to aid application profiling: Raksha[23] and Flexitaint[24] implement memory taint propagation tracking for security purposes while MemTracker[25] and HeapMon[26] detect memory access bugs. With the increasing ubiquity of multi-core architectures, proposals for dedicating one of the cores to profiling have emerged: Chen, Shimin and Falsafi[27] suggest a Log Based Architecture in which a capture is done of a program’s trace and then it is sent to an idle core for interpretation while He and Zhai[28] propose a hardware based extraction logic which is software configurable.

The main problem with having dedicated hardware for profiling is that it is not commonly available and, for now, the need of introducing such...
hardware in commodity products is not that high since traditional software profiling and debugging techniques give acceptable results. Even dedicating existing cores to profiling is not common since it involves a lot of work in getting the inter-core communication to function properly and the benefits are minimal. It would be possible to figure out the memory consumption of a process by inspecting the contents of an address space aware memory management unit’s page tables and the contents of an existing hardware stack. Fast and unintrusive access to these is required so that profiling does not interfere with their normal functioning. There exist many possible approaches to this but it is, for now, economically unsound to spend time analyzing them, given the current situation of commodity hardware. There are signs though that this is the direction we are heading towards, with the increasing number of performance counters available in today’s hardware, so perhaps in the future such dedicated hardware will not be uncommon.

### 3.1.5 Event-based Profiling

Finally, we mention profiling based on the triggering of certain events. These events can be either software or hardware and are usually provided by the environment without the possibility of modification. Software events are usually implemented as hooks into key points of an existing application, in which a profiler can insert its own code. For example, important routines involved in the processing of a network packet such as receiving and sending can provide hooks which allow monitoring the total number of sent packets or even their modification. The most common hardware events are interrupts and they can be intercepted with help of hooks provided by the operating system.

The main downside of using this type of profiling is that the events are preconfigured and adding new types of events requires heavy modification of the software or hardware platform which is not always feasible or possible. Moreover, the information passed to the hooks or callbacks might not be adequate for all but the most simple of analyses.

### 3.2 Heap Profiling

In section 2.2 we have presented the typical layout of a program after it has been loaded into memory. While in modern programs there exist a lot more sections than the ones described, the heap is usually the one where most of the allocations are done. Thus, we will not concern ourselves with the other sections because their size is pre-determined from compile time
and they do not suffer modifications during run-time. In this subchapter we will present different methods of determining the size and the point in the program where allocations on the heap are performed.

### 3.2.1 Allocation Size Profiling

The first problem we want to solve is the problem of determining the total size of all the data that exists on the heap. More specifically, we want to be able to answer one of our original questions: how much memory have the classes/modules allocated on the heap?

**Overloading memory allocation routines**

A first solution to keeping track of all the allocations that a program has done is to overload the routines that do the allocations. By doing this, we can insert our own code in the routines, code which allows us to manipulate the allocation information in any way we want. The routines which have to be overloaded are the same ones presented in section 2.3.

There are several problems with this approach, one of them related to the actual implementation of the mechanism. Overloading the routines means replacing them with our own while keeping the functionality intact. This has to be done in a way that is transparent to the running program and has very little overhead, preferably none. Different approaches exist:

- The *new* and *delete* operators can be overridden globally through language constructs provided by C++ itself. By looking at the way these two operators are implemented in the standard C++ library, one could provide an implementation that is identical but also provides additional profiling code.

- For the *malloc*, *realloc* and *free* routines, GNU libc provides hooks which allows their behaviour to be modified. These hooks are actually variables declared in malloc.h: `_malloc_hook`, `_realloc_hook`, `_free_hook`, `_memalign_hook`. All of these can point to independent routines which are called whenever the original allocation routines are called. These routines’ signature contains a caller parameter which is the return address found on the stack when the allocation routines were called, thus allowing allocation point tracking[29]. The downside with using this method is that it is GCC specific, so if other compilers are used then either a similar mechanism has to exist for them or this approach does not work.
A separate library providing implementations for all the C-level allocation routines can be used. Since \textit{new} and \textit{delete} are also using these, they will also be taken into account, thus covering the whole range. This library can then be linked with the original program in such a way that the overloaded routines are used instead of the ones provided by the standard library. This is the approach that Valgrind uses, by exporting symbols which take precedence over the ones in glibc.so\cite{30}. While it does have the benefit of being unintrusive it still is dependant on the build system, especially on the linker used.

Another solution is to provide wrappers for the allocation routines, which will be used instead of the original ones. The downside to this is that it is very intrusive since all of the original calls have to be replaced with calls to the wrappers. Tools that do this replacement automatically can be used.

Hunt and Brubacher\cite{31} classify techniques of intercepting function calls on Windows into four categories:

1. \textit{Call replacement in application source code} - All of the above, except for the one that involves providing a separate library, fit into this category.

2. \textit{Call replacement in application binary code} - By using symbolic information, call sites are identified and jump code to profiling routines can be inserted.

3. \textit{DLL redirection} - Similar to using a separate library, the internals of this technique are Windows-specific.

4. \textit{Breakpoint trapping} - By inserting a debugging breakpoint in the function we wish to intercept, we can have the debug exception handler reroute to a profiling routine. This involves a separate process (the Windows debugger) and it has the downside of suspending all application threads.

Hunt and Brubacher compare these techniques with their interception implementation and show that the overhead varies from 250\textit{ns} to 400\textit{ns} with call replacement and DLL redirection, while breakpoint trapping has an overhead on the order of microseconds. If we add to this the fact that the profiling routine itself induces overhead, along with the fact that it proves to be non-trivial to implement and sometimes even intrusive, we can conclude that overloading the memory allocation routines in order to obtain live heap information is not a viable solution.
On-demand memory tracking

We now take a different approach to keeping track of the amount of allocated memory, one which does not involve interfering with the allocation routines. To do that, we note that most of the data living on the heap is structured in some way. Whether it is stored in just a simple array of integers or more complex data structures, it has references to it which can be accessed to determine its size. The advantage of such an approach is that we control when the size is determined and thus implicitly control when the overhead of this computation is imposed. The idea is to trigger the computation of the data structure’s size on-demand, shifting the constant overhead of overloading memory allocation routines to a one-shot significantly larger overhead which could potentially be triggered during a period of low processor utilization.

The first possible way of keeping track of a data structure’s size is counter-based. This is as simple as keeping a counter which keeps track of the size that the data structure occupies, and is updated accordingly for each modification of the data structure. For example, an addNode function for a linked list would increment the variable with the size of the newly added node, while a removeNode function would decrement it in a similar manner. Naturally, more complex structures would require perhaps more counters and an even more careful accounting method, but the idea is the same: have a set of variables which accurately represent the size of the data structure at any point in time. The biggest advantage of this method is that it has very low overhead. The bulk of the accounting is spread between the methods which update the data structure and usually this involves only incrementing or decrementing the variables. When the information related to the data structure’s size is required on-demand, all there is to do is to return the variables which contain this information, making this approach very lightweight in terms of overhead. The downsides are that it is intrusive, but, more importantly, it is very hard to maintain. Experience has shown [32] that people forget to update the profiling code when the data structure is updated, or partially update the profiling code since it is spread out in many methods that have an impact on the data structure. This leads to incorrect reporters that might not even be acknowledged as incorrect until after some serious debugging.

Since the main problem with the above method was that the profiling was spread in so many places that it was hard to keep track of all of them when they needed to be updated, perhaps there is a way to aggregate all of the profiling into one place. This is the idea with traversal-based profiling. Have one method (or several, if multiple statistics are monitored) which traverses the data structure and reports its size. This does have significantly larger
overhead than the above technique, especially if the data structure is large, but it is easier to maintain. Also, let’s not forget that the idea is to trigger this traversal on-demand. There are several complicating factors with this approach, such as:

- Cycles in the data structure could lead to the same memory being counted twice.

- When using inheritance, the sub-classes must make sure not to take into account the memory of their parent classes again.

- Complex structures require complex traversals which are not trivial to implement and therefore might be difficult to maintain.

Note that by using these methods we have now lost the ability to detect memory leaks. If we would have kept track of every allocation then this extension would have been possible with some effort. However, this was never the purpose of this thesis, so memory leak detection is out of scope. Allocations which are done and then never freed and do not have a reference to them will still continue to live on the heap and will occupy space but will not be detected by the profiler. This is considered a programmer error and specialized tools for their detection do exist.

In conclusion, both of the above methods are highly intrusive, requiring access to the source code. Counter-based profiling is the lightest of the two, but the hardest to maintain, while traversal-based has a higher overhead but better maintainability. Which one should be chosen is a matter of the project’s size and priorities.

### 3.2.2 Allocation Point Profiling

The second problem we want to solve is to be able to answer two of our initial questions: who did the allocation and what lead to the allocation being done? The answer to both these questions is found in the stack trace from the moment the allocation is done.

**Manual stack traversal**

As we have seen in section 2.4, the stack is where we can find information about the call chain that led to an allocation. Accessing the stack is, unfortunately, not a straight forward endeavour, mostly because each platform has subtle differences in the way the stack is implemented, which makes accessing it a bit difficult. Some compilers provide already implemented routines which
hide away the details of the underlying architecture. One such example is GNU libc, which provides the \textit{backtrace} function. This function returns the call chain in a buffer of a given size. What it actually does behind the scenes is perform a stack walk.

A piece of software which does not want to be tied to a specific compiler should not use such compiler-provided functions but instead opt to implement its own. To give an idea of the complexity of a stack walker we present the C implementation of such a program on an x86 Linux platform.

Keeping in mind the structure of a stack frame, described in section 2.4, we need to determine two things:

- how to jump from stack frame to stack frame
- how to obtain the return address from each frame, knowing that this return address is what determines the caller

Knowing only the beginning of the stack is of no use to us since we do not know how much local data has been pushed on the stack and therefore cannot determine precisely where the return address is. In this case, we can use the \texttt{ebp} register which is commonly used to point at the beginning of the current local data. However, we know that just above the local data lie the ebp value of the caller and the return address. Thus, to jump from stack frame to stack frame we have to follow the ebp values and to get the return address we just look at the value above the ebp on the stack.

Without going into the exact implementation details, a solution that does this is shown in 3.1:

\begin{verbatim}
Listing 3.1: Simple stack walker for x86

struct frame {
    struct frame* old_fp;
    long ip;
};

struct frame *frame, *fp;
asm("movl %ebp, %0" : "=r"(frame));
fp = frame;
for( ; !(fp < frame) && !(fp < stack_bottom));
    fp = (struct frame*)((long)fp->old_fp))
{
    // Do something with the return address from fp->ip
}
\end{verbatim}
The end result is that we can obtain a list of return addresses which can then be further used to obtain the actual names of the routines forming the call chain. Inserting the above routine in every allocation point would give us a stack trace which can be used to determine the exact call chain leading to the allocation.

There are, however, several downsides to this approach. First of all it is heavily platform dependant. The above code only runs on x86, using specific GCC directives. Not only that, but it relies on the fact that the code has been compiled with frame pointers activated. Some compiler-level optimizations remove the frame pointers to reduce stack frame size and obtain a small increase in speed. Different hardware platforms may have a completely different stack frame format so the code would have to be rewritten for each compiler/platform combination, leading to something that would probably be very hard to maintain. The second problem is overhead. Attaching this code to every allocation point can lead to unnecessary overhead especially if we are not interested in the associated stack traces. A better method would be to activate the stack tracing on-demand, just for those allocations in which we are interested.

Low overhead tracepoints

The problem of low overhead tracepoints has been under discussion for a long time, especially in the context of debugging. The DTrace tool for Solaris allows probes to be inserted into a running program which have low overhead when they are disabled[12]. Such implementations have also been attempted on SPARC[33] and the LTTng project had a series of tools dedicated to tracing Linux both in userspace and kernel[34]. We will present the approach currently taken by the Linux kernel in this section.

A naive implementation of an on-demand triggerable tracepoint would just check the truth value of a flag and, based on that value, either call the tracing routine or not. It could be something as simple as the code in 3.2. There are some problems which stem from this such as the need of a data structure which keeps a list of all the available tracepoints and implements some naming scheme allowing the user to enable/disable them independently. The question is if there is some way to avoid the condition check so that a disabled tracepoint would have even lower overhead.
The idea is to keep a list of all the statically defined tracepoints. In our case, since we want all allocation points to be traceable, there will be a tracepoint for each of them. This list is built by the compiler during compilation and placed in a special section in the executable which can be accessed during runtime. At the same time, tracepoints which are disabled are replaced with nop operations. To activate a tracepoint during runtime one has to lookup its address in the tracepoint table and replace the nop instruction from that address with a jump to the place in the code which calls the tracing function. The key to having this work is special compiler support for moving code which can be jumped to, but not accessed directly, out of line[35]. The listing from 3.3 shows the way this is done in the Linux kernel, along with a typical usage scenario.
__tracepoint_ptr_##name
__attribute__((section("__tracepoints_ptrs"))) =
 &__tracepoint_##name;

if (static_branch(&__tracepoint_##name.key))
   trace(&__tracepoint_##name);

It works by inserting a nop at the label defined by 0:. In the section
called __jump_table we save the address of that label, the address of the label
to jump to when the tracepoint is activated, along with a key identifying
the tracepoint uniquely. Since the routine is typically used in a branch, and
since it will always evaluate to false, a compiler will want to remove the code
completely since it is unreachable. However, due to the jump to the Lyes
label, it is not removed completely but moved somewhere out of line thus
leaving only the nop instruction in place. We know where the code is moved
because we have saved the address of the Lyes label, thus, in order to activate
the tracepoint we have to replace the nop with a jump to that address.

This implementation shows that it is possible to achieve a tracepoint
implementation whose only significant overhead is related to the code size
of the nops and the out of line code. The downside is the same as the
stack walker’s: the implementation is platform specific. In this case it might
even be worse since the optimization that the GCC compiler does by moving
the code out of line and allowing labels inside an assembly block might not
even be possible in other compilers. Permission to modify the program’s
code during runtime is also required and this might not be allowed in secure
environments. The main issue is thus one of maintainability and of deciding
if the cost of implementing and maintaining such a solution is indeed lower
than the benefit of being able to trace call chains in key points of the program.

One final note to keep in mind is that the question we are asking is if it
is possible to implement the above and tie it into a piece of software so that
it can be used live and without damaging its performance. There are tools
which already do this sort of tracing, such as DTrace mentioned above and
even Valgrind so it is not an issue of implementation but rather performance
and maintainability.

Global stack object

The above two solutions suffer most on the maintainability side because
of their platform dependencies. The question which naturally follows is if
we can abstract away those parts into something which is independent of
the platform we are running on. The answer to this would be to keep our
own pseudo-stack (or stacks in case of multiple threads) which is globally
accessible and can be queried regarding its state at any time. We say pseudo-stack because we would only be keeping the function names in it since that is what we are interested in. To have this working, each function must call one routine at its entry point pushing its name on the stack and another at its exit point for popping. A tool which inserts these calls automatically can be created.

The global stack object thus removes the need of having a stack walker. However, invoking the object for providing the call chain still requires the tracepoints and making these platform independent leads us to the naive implementation from listing 3.2.
Chapter 4

Results and Solution

This chapter first presents a set of tests which have been run in order to determine which of the approaches described above are best suited for a solution which has to satisfy our initial constraint, that of having low overhead. A solution based on analysis of these tests is then proposed and described, along with its advantages and disadvantages.
4.1 Test Results

4.1.1 Test Infrastructure Description

In order to create a test program for the above methods we have to determine the requirements for such a program. Since we are interested in determining allocation sizes and allocation points, the test program has to provide a sufficiently diverse combination of these. Our program also has to be deterministic so that we test the methods against the same sequence of allocations. The number of allocations has to be sufficiently high so that the overhead of monitoring becomes noticeable.

In order to keep things simple and focus on the techniques and not on what the program does, the main task of our test program is to allocate a linked list whose nodes contain pointers to malloc-allocated memory whose size is controllable. In other words, we have a list of memory regions allocated with malloc. The linked list’s nodes are also heap allocated and since there is one node for each allocation we can say that the number of actual allocations is twice that we give as input to the program. We are also interested in being able to control the depth at which the allocations are made, in order to be able to determine the overhead of stack tracing as the stack increases in size.

Most programs have memory usage patterns containing a mix of allocations and deallocations. Having a test program which contains just allocations would not be representative for the majority of these patterns. Thus, we introduce the capability of deallocating some of the allocated memory through a simple counter which triggers the release of a previously allocated memory region. In the end, the core loop of the test program has the following pseudocode, where capitalized variables are given by the user:
First, we want to determine the overhead of determining the size of each allocation and compare it with the overhead of periodically sampling and traversing data structures. These are the two major approaches we can take in determining the amount of memory a specific program occupies. The final goal of allocation size monitoring is to be able to determine memory consumption on a per-module basis. Total memory consumption is not an issue, as this can be determined through other mechanisms which are usually provided by the operating system. The main problem is to have more fine grained memory reporting. For now, however, the test program is only interested in determining the total size of all the allocations we do. At this point, we are only trying to determine the overhead of obtaining the allocation data so we ignore the overhead of its utilization in finely grained memory reporting. In order to do this, we test the following scenarios:

1. **On-demand data structure traversal** - go through the linked list and use malloc_usable_size on each node and the memory region it points to.

2. **On-demand counter based monitoring** - have the linked list hold a counter representing the total allocated size, which is updated whenever a node is added or removed; access the counter whenever the total allocated size is required.

3. **GCC provided malloc hooks** - use these to insert own code which updates a global variable containing total allocated bytes.

4. **GCC aided call replacement** - write own allocation routines which do the counting and then call the existing ones to actually do the allocation.

5. **Manually defined malloc-wrappers** - use the preprocessor or just write own routines which do the counting and then call the allocation routines.

6. **Dynamically linked library containing malloc implementations** - very similar to the call replacement except it is not GCC dependent.

Second, to determine the overhead of obtaining stack traces, we test the following:
1. **GCC provided malloc hooks** - contain a parameter which gives the return address found on the stack

2. **Global stack object** - manually keep a copy of the stack on the heap and access that copy whenever we want to a stack trace

3. **Manual stack walk** - use low-level platform information about the stack’s format to perform a manual walk

4. **External library (libunwind)** - an existing library which abstracts away all the details of the stack and provides a simple way of accessing it

Running the basic program under Valgrind with the ”none” tool performs approximately 5 times worse than without Valgrind. To be more specific, the average runtime of the Valgrind run, doing 120000 allocations of 128 bytes is around 146 milliseconds while the basic run has an average runtime of 27 milliseconds. This performance ratio holds for other number of allocations and sizes. The ”none” tool does no work at all so it is a good way to measure the overhead of Valgrind’s code translation overhead. Since this overhead is significantly higher than the above mentioned approaches, we will not take Valgrind into consideration for determining allocation sizes and allocation points.

**4.1.2 Allocation Size Overhead Results**

One problem when comparing the two main methods of obtaining allocation sizes is to make sure that the results we obtain are the same so that the work can be fairly compared. Taking a closer look at these methods we observe that on-demand data structure traversal has a simple yet very important advantage over overloading the allocation routines: easy association between the data structures and their size. Our simple scenario has us incrementing a global variable which keeps track of the total amount of allocated memory so we don’t need such an association. The initial purpose was however to provide a more granular memory reporting solution. To make the comparison more fair, code that walks the entire stack manually has been inserted in the overloaded allocation routines. This information would then theoretically be used to provide an accurate location of where the allocation was made.

In figure 4.1 we can see that the overhead of overloading is lower than the traversal’s when we only increment/decrement the global variable. This is explained by the fact that the traversal involves sequential access through each list element which in turn generates extra page faults and cache misses.
The overhead of these additional memory accesses is thus higher at this point than the work done inside the overloaded allocation routines. In figure 4.2 however, we can see that this is no longer true when we add the stack walking. The conclusion to be drawn from this is that the actual mechanisms used to obtain the allocation size information are not the ones inducing the overhead but rather the work performed inside these mechanisms. Since a lot of work needs to be done in the overloaded allocation routines in order to correctly identify the place where the allocation was made, they do perform worse than the traversal techniques.

4.1.3 Allocation Point Overhead Results

In figure 4.3 the time overhead of the allocation point determination is shown. While it may seem that the malloc hooks provide the best solution, it has to be mentioned that they only provide the caller of the malloc routine which makes them useless for practical purposes. For the hooks to provide the same amount of information as the other methods, they would have to be augmented with a similar stack walking routine which would put them on the same overhead level as the manual walking method.

Comparing the global stack object with manual stack walking we can see that the former appears to be significantly better. There is one small catch
related to the global stack object and that is its behaviour in a multithreaded environment. Multiple buffers are required, one for each thread. Additional code has to be added to check which thread has called the current function, in order to determine in which buffer the trace will be placed. Another disadvantage is that every method has to be augmented at entry and exit points with calls to the object’s logging method. This could be done before compile time, by an automatic script. That being said, calls are expensive so inlining might be a solution, at the expense of increased code size.

Finally, using libunwind has such a high overhead that when the results get added to the graph, the other three methods’ plot points degenerate into a line. It has thus been omitted from the graph but the method does have its merits. Such a library represents the most portable way of implementing allocation point tracing. Being able to ignore all the low-level details and use a common interface for all of them represents a huge benefit which might make this method suitable if the performance hit is acceptable to the application.

All of the above methods have some overhead, so the question is if we can minimize that overhead no matter what method we choose. The first observation we can make is that in the method analysis we have tried to obtain stack traces at each and every allocation the program makes. While this does provide a complete view of an application’s memory usage char-

Figure 4.2: Allocation size time overhead compared to the basic scenario when logging is done
actoristics it is unnecessary for analysis targeting memory spikes and high memory consumption, which is what we are interested in. The typical usage scenario of the profiler we want is the following:

1. Have a complete view of the memory consumption of different modules in the application
2. Observe one or more modules which show an unusually high memory consumption
3. Trigger more in-depth analysis of those modules by showing the most frequently called allocations done inside the module
4. Enable stack trace logging only for those allocations which are frequently called

By doing the above we can see the call chain which leads to frequent allocations and identify points which can be optimized for better performance.

This is the part where the low-overhead tracepoints come into play. Each allocation routine can have such a tracepoint attached which is disabled by default, by being a noop. The tracepoint, when enabled, does a jump to a routine which either does a simple call count or a full-fledged stack trace,
depending on a flag. The enabling of these tracepoints is done entirely on-demand, thus avoiding the overhead of having all the allocation routines do stack trace logging. Combining the tracepoints with any of the allocation point methods above, leads to a lightweight solution that is, however, platform specific and not easily implementable.

4.1.4 Other Issues

There are other issues which the tests do not tackle, yet they are important for a complete solution:

1. **Information storage and analysis** - The test programs use a circular, fixed size buffer for storing the stack traces up to a certain depth. This solution has to be extended to add allocation size information and to allow grouping of stack traces. Two allocations made from the same point have the same stack trace and thus the size should be modified accordingly. Exactly how much of the stack trace is to be compared for equality is another discussion. A shallow comparison of just a couple of stack frames from the trace might not be too useful if the allocations are done in a number of function calls larger than the analysis depth. A specialized allocator might be such an implementation and a shallow analysis would just reveal how the allocator works when in fact we are interested in determining who called the allocator.

Another related problem is where this analysis should be performed. If it is performed at the allocation points we run into the risk of very large overhead. Another approach is to simply add everything into the data structure which holds the traces and let the visualisation component handle the analysis. This way, it gets triggered on-demand and incurs less overhead.

We can also ask ourselves how long should the information be stored? Do we just keep the N most recent stack traces or do we keep all the traces since the program has started? It depends on what type of analysis is done. If we want to be able to visually track the memory consumption and just make sure that everything is in acceptable parameters, a shorter history can be kept. This, however, has the danger of missing memory spikes, where memory consumption increases rapidly but it is not noticed. Here, a longer history is required, to be used by analysis tools and not just for visual inspection. Naturally, the longer the history, the more memory it consumes and writing information to disk very often has a very large time penalty since I/O
is inherently slow. A middle ground is required, where the data is dumped periodically on disk or during low activity times.

2. **Concurrency** - Different threads use different stacks and the analysis needs to take this into consideration. Moreover, the data structure holding the traces now needs to be locked to protect it from simultaneous access from multiple threads trying to push stack traces. The visualiser also needs to fetch data from the data structure and thus has to lock it completely to make sure that it is not modified while being read. This scenario can be described as a multiple writers, one reader situation.

### 4.2 Solution Description

By analyzing the test results, we can see that the best solutions for allocation size logging seem to be either the on-demand traversal or the on-demand counter techniques. As mentioned before, the counter technique has the disadvantage of being slightly harder to maintain. The underlying reason for this is that the responsibility of updating the counter variable is unavoidably split between all the functions which modify the associated data structure in some way. That means that when one of these functions changes or a new one is added, additional care has to be taken so that the counter correctly reflects the changes that the functions bring. Thus, the on-demand traversal technique is preferred over the counter technique.

There is, however, a use for the counter technique too. In cases where the data structure which is profiled has been present in the code base for a long time and when it does not change that often, it might make sense to use this. During the implementation of this solution there were areas of code which already had such data structures, along with an already implemented counter. If splitting the counter into multiple reports is not desired (for example, it might not be interesting to see which nodes from a list belong to whom) then this technique is definitely usable. The decision of whether or not to use it has to be made on a case-by-case basis, taking into account the stability of that code area, developing time to implement traversal and the potential benefits brought by splitting the report.

The exact method through which the traversal is implemented is by calling malloc_usable_size on all of the data structure’s pointers. A simple example is given in listing 4.2, where a structure is defined, along with a memory reporter and it is assumed that all of the structure has been allocated on the heap using traditional allocation routines such as malloc and new.
Listing 4.2: Traversal example

```c
struct node {
    int *field1; // array of ints
    int *field2; // array of ints
    struct node *next;
};

struct node *List;

int reporter_function {
    int size = 0;
    struct node *iterator = List;
    size += malloc_usable_size(List);
    while (iterator != NULL) {
        size += malloc_usable_size(iterator->field1);
        size += malloc_usable_size(iterator->field2);
        size += malloc_usable_size(iterator->next);
        iterator = iterator->next;
    }
    return size;
}
```

Moving on to the selection of an allocation point technique, according to the graphs, it appears that the global stack object approach is the best. However, there is another technique mentioned, which is not shown in the graphs. That technique is the low overhead tracepoints and the reason it is not mentioned in the graphs is because their overhead is very close to zero when they are not enabled, and when they are enabled the overhead is dependent on how the tracing is done.

Another solution for the allocation points is one that is available at the moment only for a few platforms: DTrace. In the same way that the low overhead tracepoints insert nops at compile time so that they can later be used to jump to tracing routines, DTrace actually overwrites instructions during runtime with jumps to "trampolines". These contain:

- The original overwritten instruction.
- A jump to the desired tracing routine.
- A way of returning to the instruction following the overwritten one.

The key difference here is the fact that DTrace does this during runtime. A process can already be running and DTrace can attach to it and use the symbol table to insert jumps to tracing points at desired locations.

The final piece of the solution is Valgrind. In response to the question: "How do we know when we have finished profiling the application?", Valgrind
can provide the answer. One of its tools is used to intercept every allocation made in the program and to group those allocations which have similar stack traces. The results are sorted in order of the number of allocated bytes and the information given is the exact line of code which has performed the allocation along with a stack trace. This information can be used by the developer implementing the profiler to identify the next code area which allocates the largest amount of memory. That area is a good candidate for continuing the profiler implementation.

In addition to that, Valgrind can also give us information about the total amount of memory allocated by a process during a run. By annotating the areas of memory we are profiling with specific code, we can inform Valgrind that we have profiled that memory and thus it can ignore it when it gives us the results. The Valgrind tool used for this keeps a log of all the allocations performed. Whenever we traverse a data structure, we annotate the allocations it has done (to which naturally, it has pointers) with code that tells Valgrind that "this allocation has been taken into account, report it no more". In the end, we can get the total amount of memory the program allocates and the total amount of memory we have marked as "seen" through profiling. When these two become the same, then we can consider that we have profiled all the memory allocated by the program.

Considering example 4.2, the annotation implies replacing all calls to malloc_usable_size with a call to either a routine or a macro which calls both malloc_usable_size and the call to Valgrind, similar to listing 4.3.

Listing 4.3: Valgrind annotation

```c
int report_memory(void *ptr) {
    notify_valgrind_that_allocation_was_reported(ptr);
    return malloc_usable_size(ptr);
}

int reporter_function {
    [...]  
    while (iterator != NULL) {
        size += report_memory(iterator->field1);
        size += report_memory(iterator->field2);
        size += report_memory(iterator->next);
        iterator = iterator->next;
    }
    [...]  
}
```

Since running a process under Valgrind has high overhead, this run is done only to determine how much of the memory still has to be profiled.
Thus, Valgrind acts as a temporary guide to implement a better low-overhead profiler.

The process connecting all of these together and describing the final solution is thus the following:

1. Check, using Valgrind, and see if the memory covered by the profiler is the same as the one reported by the Valgrind tool. If they are the same, the profiler can be considered done. If not, go to step 2.

2. Identify, using Valgrind, a code area which does a lot of memory allocations. The information available at this point is that line X has allocated memory along with the path leading to that allocation.

3. Identify by looking at the source code which module or class or other structural entity owns that line of code.

4. Extend the profiler to take into account the memory allocated by that module.

5. (Optional) Enable tracepoints after the module’s allocation points to get stack traces in order to determine which code paths are responsible for the majority of allocations done in the module. Use this information to optimize code.

6. Go to step 1.
Chapter 5

Conclusion

There are already multiple solutions for memory profiling, but the one thing that they all have in common is that they have high enough overhead such that using them in normal builds significantly slows down the software. The future does hold some promise though, with the advent of tools such as DTrace which attempt low overhead profiling. Hopefully at some point these tools will become mature and ubiquitous enough that they can be used to build extremely fast profilers.

As it stands now, this thesis describes an attempt to implement a low-overhead profiler which runs in the same process as the target software. A partial implementation has been successful, providing a proof-of-concept and acting as a reference and starting point for future attempts. The implementation has brought with it some insight into the actual difficulties of implementing such a profiler.

First of all, the presented process is highly intrusive. Source code access is required and, perhaps more important, source code familiarity is crucial. In order to create a consistent hierarchy of modules or classes whose memory is reported, one has to have knowledge about the software’s entire architecture from a top-down perspective. Even more, the most important thing to know is how everything is tied together.

An ideal situation would be that the software has a perfectly modular architecture, where calls between modules are few and they share no data. This is, however, rarely true in practice. One of the biggest challenges when implementing the solution was finding out which data is shared between classes and to decide to which class the memory is ”assigned” when the memory reporting is done. This cannot be done arbitrarily in most cases but the overall architecture has to be taken into consideration and decisions must have a reason. Shared data is thus a problem and while double reports to Valgrind can be solved, reporting the same memory twice to the user might
give the impression that the software uses more memory than it actually does.

The second major issue was that reporting memory with malloc_usable_size is limited to memory allocated by the default allocator. Special allocators require special handling both on the reporting side and on the Valgrind tool side. To take an example, imagine an allocator which handles small objects. Such an allocator would request one large chunk from the default allocator or mmap that chunk itself and then allocate small pieces from that chunk. The only allocation that is seen from both malloc_usable_size’s and Valgrind’s view is the initial large one. The rest are invisible to them. Thus, special techniques have to be implemented for these cases, and these become more complex as the complexity of the allocators themselves increases. Unfortunately, such allocators are not uncommon in modern software since the speed gain brought by them is significant enough to warrant their usage.

Finally, related to the optional step in the process involving the trace-points, only a separate implementation has been successful. The solution does work, but integrating it into a larger project might prove tricky, especially due to the custom linker scripts and the need for runtime code modifications requiring root access. Another small disadvantage of having tracepoints is the fact that the source code gets littered with them and they can become a nuisance so perhaps special build scripts might have to be built to offer more readable versions of the source code. The tracepoints can be packed into a define containing the allocations in our case, but if we want to trace the beginning or end of functions then we have to add those manually, thus cluttering the code. DTrace does not have this problem so it will provide a better solution for the optional step.

In conclusion, when implementing a memory profiler the trade-off is between intrusiveness and speed. The solution provided here is very intrusive and very time consuming to implement but it is very fast since the memory profiling is triggered on demand. Future tools promise better approaches, current ones are constantly being worked on, all of this coupled with increasing processor speed, might make external tools viable in the future. For now, if speed is a critical constraint, intrusive solutions are the most viable.
Bibliography


[27] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix,


[35] Richard Henderson. Beginning of discussions to introduce asm goto into GCC.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Andrei Faur]