9. Massif: a heap profiler

Table of Contents

To use this tool, you must specify `--tool=massif` on the Valgrind command line.

9.1. Overview

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.
- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

Also, there are certain space leaks that aren't detected by traditional leak-checkers, such as Memcheck's. That's because the memory isn't ever actually lost -- a pointer remains to it -- but it's not in use. Programs that have leaks like this can unnecessarily increase the amount of memory they are using over time. Massif can help identify these leaks.

Importantly, Massif tells you not only how much heap memory your program is using, it also gives very detailed information that indicates which parts of your program are responsible for allocating the heap memory.

Massif also provides Execution Trees memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

9.2. Using Massif and ms_print

First off, as for the other Valgrind tools, you should compile with debugging info (the `-g` option). It shouldn't matter much what optimisation level you compile your program with, as this is unlikely to affect the heap memory usage.

Then, you need to run Massif itself to gather the profiling information, and then run ms_print to present it in a readable way.

9.2.1. An Example Program

An example will make things clear. Consider the following C program (annotated with line numbers) which allocates a number of different blocks on the heap.

```
1     #include <stdlib.h>
2
3     void g(void)
4     {
5        malloc(4000);
6     }
7
8     void f(void)
9     {
10       malloc(2000);
11       g();
12    }
13
14    int main(void)
15    {
16       int i;
17       int* a[10];
18
19       for (i = 0; i < 10; i++) {
20          a[i] = malloc(1000);
21       }
22
```

```
23        f();
24
25        g();
26
27        for (i = 0; i < 10; i++) {
28           free(a[i]);
29        }
30
31        return 0;
32     }
```

9.2.2. Running Massif

To gather heap profiling information about the program `prog`, type:

```
valgrind --tool=massif prog
```

The program will execute (slowly). Upon completion, no summary statistics are printed to Valgrind's commentary; all of Massif's profiling data is written to a file. By default, this file is called `massif.out.<pid>`, where `<pid>` is the process ID, although this filename can be changed with the `--massif-out-file` option.

9.2.3. Running ms_print

To see the information gathered by Massif in an easy-to-read form, use ms_print. If the output file's name is `massif.out.12345`, type:

```
ms_print massif.out.12345
```

ms_print will produce (a) a graph showing the memory consumption over the program's execution, and (b) detailed information about the responsible allocation sites at various points in the program, including the point of peak memory allocation. The use of a separate script for presenting the results is deliberate: it separates the data gathering from its presentation, and means that new methods of presenting the data can be added in the future.

9.2.4. The Output Preamble

After running this program under Massif, the first part of ms_print's output contains a preamble which just states how the program, Massif and ms_print were each invoked:

```
--------------------------------------------------------------------------------
Command:            example
Massif arguments:   (none)
ms_print arguments: massif.out.12797
--------------------------------------------------------------------------------
```

9.2.5. The Output Graph

The next part is the graph that shows how memory consumption occurred as the program executed:

```
    KB
19.63^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
   0 +----------------------------------------------------------------------->ki    0                                                                  113.4


Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

Why is most of the graph empty, with only a couple of bars at the very end? By default, Massif uses "instructions executed" as the unit of time. For very short-run programs such as the example, most of the executed instructions involve the loading and dynamic linking of the program. The execution of `main` (and thus the heap allocations) only occur at the very end. For a short-running program like this, we can use the `--time-unit=B` option to specify that we want the time unit to instead be the number of bytes allocated/deallocated on the heap and stack(s).

If we re-run the program under Massif with this option, and then re-run ms_print, we get this more useful graph:

```
19.63^                                       ###
     |                                       #
     |                                       #  ::
     |                                       #  : :::
     |                                  :::::::::#  : :  ::
     |                                  :       #  : :  : ::
     |                                  :       #  : :  : : :::
     |                                  :       #  : :  : : :  ::
     |                           ::::::::::     #  : :  : : :  : :::
     |                           :      :       #  : :  : : :  : :  ::
     |                        :::::     :       #  : :  : : :  : :  : ::
     |                     @@@:   :     :       #  : :  : : :  : :  : : @
```

```
     |                ::@  :   :        :       #  : :  : : :  : :  : : @
     |              :::: @  :   :        :       #  : :  : : :  : :  : : @
     |            :::  : @  :   :        :       #  : :  : : :  : :  : : @
     |          ::: :  : @  :   :        :       #  : :  : : :  : :  : : @
     |        :::: : :  : @  :   :        :       #  : :  : : :  : :  : : @
     |      :::  : : :  : @  :   :        :       #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :        :       #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :        :       #  : :  : : :  : :  : : @
   0 +-------------------------------------------------------------------->KB     0                                                                    29.48

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

The size of the graph can be changed with ms_print's `--x` and `--y` options. Each vertical bar represents a snapshot, i.e. a measurement of the memory usage at a certain point in time. If the next snapshot is more than one column away, a horizontal line of characters is drawn from the top of the snapshot to just before the next snapshot column. The text at the bottom show that 25 snapshots were taken for this program, which is one per heap allocation/deallocation, plus a couple of extras. Massif starts by taking snapshots for every heap allocation/deallocation, but as a program runs for longer, it takes snapshots less frequently. It also discards older snapshots as the program goes on; when it reaches the maximum number of snapshots (100 by default, although changeable with the `--max-snapshots` option) half of them are deleted. This means that a reasonable number of snapshots are always maintained.

Most snapshots are *normal*, and only basic information is recorded for them. Normal snapshots are represented in the graph by bars consisting of ':' characters.

Some snapshots are *detailed*. Information about where allocations happened are recorded for these snapshots, as we will see shortly. Detailed snapshots are represented in the graph by bars consisting of '@' characters. The text at the bottom show that 3 detailed snapshots were taken for this program (snapshots 9, 14 and 24). By default, every 10th snapshot is detailed, although this can be changed via the `--detailed-freq` option.

Finally, there is at most one *peak* snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of '#' characters. The text at the bottom shows that snapshot 14 was the peak.

Massif's determination of when the peak occurred can be wrong, for two reasons.

- Peak snapshots are only ever taken after a deallocation happens. This avoids lots of unnecessary peak snapshot recordings (imagine what happens if your program allocates a lot of heap blocks in succession, hitting a new peak every time). But it means that if your program never deallocates any blocks, no peak will be recorded. It also means that if your program does deallocate blocks but later allocates to a higher peak without subsequently deallocating, the reported peak will be too low.
- Even with this behaviour, recording the peak accurately is slow. So by default Massif records a peak whose size is within 1% of the size of the true peak. This inaccuracy in the peak measurement can be changed with the `--peak-inaccuracy` option.

The following graph is from an execution of Konqueror, the KDE web browser. It shows what graphs for larger programs look like.

```
    MB
3.952^                                                                #
     |                                                               @#:
     |                                                             :@@#:
     |                                                          @@:::@@#:
     |                                                          @ :: :@@#::
     |                                                        @@@ :: :@@#::
     |                                                     @@:@@@ :: :@@#::
     |                                                  :::@ :@@@ :: :@@#::
     |                                                  : :@ :@@@ :: :@@#::
     |                                               :@: :@ :@@@ :: :@@#::
     |                                             @@:@: :@ :@@@ :: :@@#:::
     |                         :       ::         ::@@:@: :@ :@@@ :: :@@#:::
     |                       :@@:    ::::: ::::@@@:::@@:@: :@ :@@@ :: :@@#:::
     |                    ::::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                  @: ::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                  @: ::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                  @: ::@@:::::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                ::@@@: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |             :::::@ @: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |           @@:::::@ @: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
   0 +-------------------------------------------------------------------->Mi
     0                                                                   626.4

Number of snapshots: 63
 Detailed snapshots: [3, 4, 10, 11, 15, 16, 29, 33, 34, 36, 39, 41,
                      42, 43, 44, 49, 50, 51, 53, 55, 56, 57 (peak)]
```

Note that the larger size units are KB, MB, GB, etc. As is typical for memory measurements, these are based on a multiplier of 1024, rather than the standard SI multiplier of 1000. Strictly speaking, they should be written KiB, MiB, GiB, etc.

9.2.6. The Snapshot Details

Returning to our example, the graph is followed by the detailed information for each snapshot. The first nine snapshots are normal, so only a small amount of information is recorded for each one:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0              0            0
  1          1,008            1,008          1,000              8            0
  2          2,016            2,016          2,000             16            0
  3          3,024            3,024          3,000             24            0
  4          4,032            4,032          4,000             32            0
  5          5,040            5,040          5,000             40            0
  6          6,048            6,048          6,000             48            0
  7          7,056            7,056          7,000             56            0
  8          8,064            8,064          8,000             64            0
```

Each normal snapshot records several things.

- Its number.
- The time it was taken. In this case, the time unit is bytes, due to the use of `--time-unit=B`.
- The total memory consumption at that point.
- The number of useful heap bytes allocated at that point. This reflects the number of bytes asked for by the program.
- The number of extra heap bytes allocated at that point. This reflects the number of bytes allocated in excess of what the program asked for. There are two sources of extra heap bytes.

  First, every heap block has administrative bytes associated with it. The exact number of administrative bytes depends on the details of the allocator. By default Massif assumes 8 bytes per block, as can be seen from the example, but this number can be changed via the `--heap-admin` option.

  Second, allocators often round up the number of bytes asked for to a larger number, usually 8 or 16. This is required to ensure that elements within the block are suitably aligned. If N bytes are asked for, Massif rounds N up to the nearest multiple of the value specified by the `--alignment` option.
- The size of the stack(s). By default, stack profiling is off as it slows Massif down greatly. Therefore, the stack column is zero in the example. Stack profiling can be turned on with the `--stacks=yes` option.

The next snapshot is detailed. As well as the basic counts, it gives an allocation tree which indicates exactly which pieces of code were responsible for allocating heap memory:

```
  9          9,072           9,072           9,000           72           0
99.21% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.21% (9,000B) 0x804841A: main (example.c:20)
```

The allocation tree can be read from the top down. The first line indicates all heap allocation functions such as `malloc` and C++ `new`. All heap allocations go through these functions, and so all 9,000 useful bytes (which is 99.21% of all allocated bytes) go through them. But how were `malloc` and new called? At this point, every allocation so far has been due to line 20 inside `main`, hence the second line in the tree. The `->` indicates that main (line 20) called `malloc`.

Let's see what the subsequent output shows happened next:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,080          10,080          10,000           80           0
 11         12,088          12,088          12,000           88           0
 12         16,096          16,096          16,000           96           0
 13         20,104          20,104          20,000          104           0
 14         20,104          20,104          20,000          104           0
99.48% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.74% (10,000B) 0x804841A: main (example.c:20)
|
->39.79% (8,000B) 0x80483C2: g (example.c:5)
| ->19.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->19.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->19.90% (4,000B) 0x8048436: main (example.c:25)
|
->09.95% (2,000B) 0x80483DA: f (example.c:10)
  ->09.95% (2,000B) 0x8048431: main (example.c:23)
```

The first four snapshots are similar to the previous ones. But then the global allocation peak is reached, and a detailed snapshot (number 14) is taken. Its allocation tree shows that 20,000B of useful heap memory has been allocated, and the lines and arrows indicate that this is from three different code locations: line 20, which is responsible for 10,000B (49.74%); line 5, which is responsible for 8,000B (39.79%); and line 10, which is responsible for 2,000B (9.95%).

We can then drill down further in the allocation tree. For example, of the 8,000B asked for by line 5, half of it was due to a call from line 11, and half was due to a call from line 25.

In short, Massif collates the stack trace of every single allocation point in the program into a single tree, which gives a complete picture at a particular point in time of how and why all heap memory was allocated.

Note that the tree entries correspond not to functions, but to individual code locations. For example, if function `A` calls `malloc`, and function `B` calls `A` twice, once on line 10 and once on line 11, then the two calls will result in two distinct stack traces in the tree. In contrast, if `B` calls `A` repeatedly from line 15 (e.g. due to a loop), then each of those calls will be represented by the same stack trace in the tree.

Note also that each tree entry with children in the example satisfies an invariant: the entry's size is equal to the sum of its children's sizes. For example, the first entry has size 20,000B, and its children have sizes 10,000B, 8,000B, and 2,000B. In general, this invariant almost always holds. However, in rare circumstances stack traces can be malformed, in which case a stack trace can be a sub-trace of another stack trace. This means that some entries in the tree may not satisfy the invariant -- the entry's size will be greater than the sum of its children's sizes. This is not a big problem, but could make the results confusing. Massif can sometimes detect when this happens; if it does, it issues a warning:

```
Warning: Malformed stack trace detected.  In Massif's output,
         the size of an entry's child entries may not sum up
         to the entry's size as they normally do.
```

However, Massif does not detect and warn about every such occurrence. Fortunately, malformed stack traces are rare in practice.

Returning now to ms_print's output, the final part is similar:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15         21,112          19,096          19,000           96           0
 16         22,120          18,088          18,000           88           0
 17         23,128          17,080          17,000           80           0
 18         24,136          16,072          16,000           72           0
 19         25,144          15,064          15,000           64           0
 20         26,152          14,056          14,000           56           0
 21         27,160          13,048          13,000           48           0
 22         28,168          12,040          12,000           40           0
 23         29,176          11,032          11,000           32           0
 24         30,184          10,024          10,000           24           0
```

```
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x80483C2: g (example.c:5)
| ->39.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->39.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->39.90% (4,000B) 0x8048436: main (example.c:25)
|
->19.95% (2,000B) 0x80483DA: f (example.c:10)
| ->19.95% (2,000B) 0x8048431: main (example.c:23)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

The final detailed snapshot shows how the heap looked at termination. The 00.00% entry represents the code locations for which memory was allocated and then freed (line 20 in this case, the memory for which was freed on line 28). However, no code location details are given for this entry; by default, Massif only records the details for code locations responsible for more than 1% of useful memory bytes, and ms_print likewise only prints the details for code locations responsible for more than 1%. The entries that do not meet this threshold are aggregated. This avoids filling up the output with large numbers of unimportant entries. The thresholds can be changed with the `--threshold` option that both Massif and ms_print support.

9.2.7. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file format string (controlled by `--massif-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by ms_print.

9.2.8. Measuring All Memory in a Process

It is worth emphasising that by default Massif measures only heap memory, i.e. memory allocated with `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]`, and a few other, similar functions. (And it can optionally measure stack memory, of course.) This means it does *not* directly measure memory allocated with lower-level system calls such as `mmap`, `mremap`, and `brk`.

Heap allocation functions such as `malloc` are built on top of these system calls. For example, when needed, an allocator will typically call `mmap` to allocate a large chunk of memory, and then hand over pieces of that memory chunk to the client program in response to calls to `malloc` et al. Massif directly measures only these higher-level `malloc` et al calls, not the lower-level system calls.

Furthermore, a client program may use these lower-level system calls directly to allocate memory. By default, Massif does not measure these. Nor does it measure the size of code, data and BSS segments. Therefore, the numbers reported by Massif may be significantly smaller than those reported by tools such as `top` that measure a program's total size in memory.

However, if you wish to measure *all* the memory used by your program, you can use the `--pages-as-heap=yes`. When this option is enabled, Massif's normal heap block profiling is replaced by lower-level page profiling. Every page allocated via `mmap` and similar system calls is treated as a distinct block. This means that code, data and BSS segments are all measured, as they are just memory pages. Even the stack is measured, since it is ultimately allocated (and extended when necessary) via `mmap`; for this reason `--stacks=yes` is not allowed in conjunction with `--pages-as-heap=yes`.

After `--pages-as-heap=yes` is used, ms_print's output is mostly unchanged. One difference is that the start of each detailed snapshot says:

```
(page allocation syscalls) mmap/mremap/brk, --alloc-fns, etc.
```

instead of the usual:

```
(heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

The stack traces in the output may be more difficult to read, and interpreting them may require some detailed understanding of the lower levels of a program like the memory allocators. But for some programs having the full information about memory usage can be very useful.

9.2.9. Acting on Massif's Information

Massif's information is generally fairly easy to act upon. The obvious place to start looking is the peak snapshot.

It can also be useful to look at the overall shape of the graph, to see if memory usage climbs and falls as you expect; spikes in the graph might be worth investigating.

The detailed snapshots can get quite large. It is worth viewing them in a very wide window. It's also a good idea to view them with a text editor. That makes it easy to scroll up and down while keeping the cursor in a particular column, which makes following the allocation chains easier.

9.3. Using massif-visualizer

massif-visualizer is a graphical viewer for Massif data that is often easier to use than ms_print. massif-visualizer is not shipped within Valgrind, but is available in various places online.

9.4. Massif Command-line Options

Massif-specific command-line options are:

`--heap=<yes|no> [default: yes]`

Specifies whether heap profiling should be done.

`--heap-admin=<size> [default: 8]`

If heap profiling is enabled, gives the number of administrative bytes per block to use. This should be an estimate of the average, since it may vary. For example, the allocator used by glibc on Linux requires somewhere between 4 to 15 bytes per block, depending on various factors. That allocator also requires admin space for freed blocks, but Massif cannot account for this.

`--stacks=<yes|no> [default: no]`

Specifies whether stack profiling should be done. This option slows Massif down greatly, and so is off by default. Note that Massif assumes that the main stack has size zero at start-up. This is not true, but doing otherwise accurately is difficult. Furthermore, starting at zero better indicates the size of the part of the main stack that a user program actually has control over.

`--pages-as-heap=<yes|no> [default: no]`

Tells Massif to profile memory at the page level rather than at the malloc'd block level. See above for details.

`--depth=<number> [default: 30]`

Maximum depth of the allocation trees recorded for detailed snapshots. Increasing it will make Massif run somewhat more slowly, use more memory, and produce bigger output files.

`--alloc-fn=<name>`

Functions specified with this option will be treated as though they were a heap allocation function such as `malloc`. This is useful for functions that are wrappers to `malloc` or `new`, which can fill up the allocation trees with uninteresting information. This option can be specified multiple times on the command line, to name multiple functions.

Note that the named function will only be treated this way if it is the top entry in a stack trace, or just below another function treated this way. For example, if you have a function `malloc1` that wraps `malloc`, and `malloc2` that wraps `malloc1`, just specifying `--alloc-fn=malloc2` will have no effect. You need to specify `--alloc-fn=malloc1` as well. This is a little inconvenient, but the reason is that checking for allocation functions is slow, and it saves a lot of time if Massif can stop looking through the stack trace entries as soon as it finds one that doesn't match rather than having to continue through all the entries.

Note that C++ names are demangled. Note also that overloaded C++ names must be written in full. Single quotes may be necessary to prevent the shell from breaking them up. For example:

```
--alloc-fn='operator new(unsigned, std::nothrow_t const&)'
```

`--ignore-fn=<name>`

Any direct heap allocation (i.e. a call to `malloc`, `new`, etc, or a call to a function named by an `--alloc-fn` option) that occurs in a function specified by this option will be ignored. This is mostly useful for testing purposes. This option can be specified multiple times on the command line, to name multiple functions.

Any `realloc` of an ignored block will also be ignored, even if the `realloc` call does not occur in an ignored function. This avoids the possibility of negative heap sizes if ignored blocks are shrunk with `realloc`.

The rules for writing C++ function names are the same as for `--alloc-fn` above.

`--threshold=<m.n> [default: 1.0]`

The significance threshold for heap allocations, as a percentage of total memory size. Allocation tree entries that account for less than this will be aggregated. Note that this should be specified in tandem with ms_print's option of the same name.

`--peak-inaccuracy=<m.n> [default: 1.0]`

Massif does not necessarily record the actual global memory allocation peak; by default it records a peak only when the global memory allocation size exceeds the previous peak by at least 1.0%. This is because there can be many local allocation peaks along the way, and doing a detailed snapshot for every one would be expensive and wasteful, as all but one of them will be later discarded. This inaccuracy can be changed (even to 0.0%) via this option, but Massif will run drastically slower as the number approaches zero.

`--time-unit=<i|ms|B> [default: i]`

The time unit used for the profiling. There are three possibilities: instructions executed (i), which is good for most cases; real (wallclock) time (ms, i.e. milliseconds), which is sometimes useful; and bytes allocated/deallocated on the heap and/or stack (B), which is useful for very short-run programs, and for testing purposes, because it is the most reproducible across different machines.

`--detailed-freq=<n> [default: 10]`

Frequency of detailed snapshots. With `--detailed-freq=1`, every snapshot is detailed.

`--max-snapshots=<n> [default: 100]`

The maximum number of snapshots recorded. If set to N, for all programs except very short-running ones, the final number of snapshots will be between N/2 and N.

`--massif-out-file=<file> [default: massif.out.%p]`

Write the profile data to `file` rather than to the default output file, `massif.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

9.5. Massif Monitor Commands

The Massif tool provides monitor commands handled by the Valgrind gdbserver (see Monitor command handling by the Valgrind gdbserver).

- `snapshot [<filename>]` requests to take a snapshot and save it in the given <filename> (default massif.vgdb.out).
- `detailed_snapshot [<filename>]` requests to take a detailed snapshot and save it in the given <filename> (default massif.vgdb.out).
- `all_snapshots [<filename>]` requests to take all captured snapshots so far and save them in the given <filename> (default massif.vgdb.out).
- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Massif tool to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

9.6. Massif Client Requests

Massif does not have a `massif.h` file, but it does implement two of the core client requests: `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK`; they are described in The Client Request mechanism.

9.7. ms_print Command-line Options

ms_print's options are:

`-h --help`

Show the help message.

`--version`

Show the version number.

`--threshold=<m.n> [default: 1.0]`

Same as Massif's `--threshold` option, but applied after profiling rather than during.

`--x=<4..1000> [default: 72]`

Width of the graph, in columns.

`--y=<4..1000> [default: 20]`

Height of the graph, in rows.

9.8. Massif's Output File Format

Massif's file format is plain text (i.e. not binary) and deliberately easy to read for both humans and machines. Nonetheless, the exact format is not described here. This is because the format is currently very Massif-specific. In the future we hope to make the format more general, and thus suitable for possible use with other tools. Once this has been done, the format will be documented here.

<< 8. DRD: a thread error detector | Up | Home | 10. DHAT: a dynamic heap analysis tool >>