

## Qnx Corner Article



### Memory Profiling using QNX Momentics IDE 4

#### Summary

This article describes various techniques for embedded software memory profiling, using tools from the QNX Momentics IDE 4 tool set.

**By Elena Laskavaia, QNX Software Systems**

Aug 31, 2007

---

The term **memory profiling** refers to a wide range of application testing tasks related to computer memory, such as identifying memory corruption, memory leaks and optimizing memory usage. The QNX Momentics IDE, which is part of QNX Momentics 6.3.2, includes tools to assist you with all of these tasks. However, this article focuses on the optimization of memory usage for better performance and smaller memory footprint. Memory efficiency is particularly critical for embedded software, where memory resources are very limited, especially with absence of swapping, and the need for processes that run continuously.

**Note:** This article assumes that you have basic knowledge of the QNX Momentics IDE (the Eclipse based Integrated Development Environment), and that you can edit, compile, and run C/C++ applications on target hosts running the QNX Neutrino RTOS.

## Process Memory

Typically, virtual memory occupied by a process can be separated into the following categories:

- **Code** — Contains the executable code for a process and the code for the shared libraries. If more than one process uses the same library, then the virtual segment containing its code will be mapped to the same physical segment (i.e., shared between processes).
- **Data** — Contains a process data segment and the data segments for the shared libraries. This type of memory is usually referred to as static memory.
- **Stack** — This segment contains memory required for function stacks (one stack for each thread).
- **Heap** — This segment contains all memory dynamically allocated by a process.
- **Shared Heap** — Contains other types of memory allocation, such as shared memory and mapped memory for a process.

**Note:** It is important to know how much memory each individual process uses, otherwise you can spend considerable time trying to optimize the heap (i.e., if a process uses only 5% of the total process memory, is it unlikely to return any noticeable result). Techniques for optimizing a particular type of memory are also dramatically different.

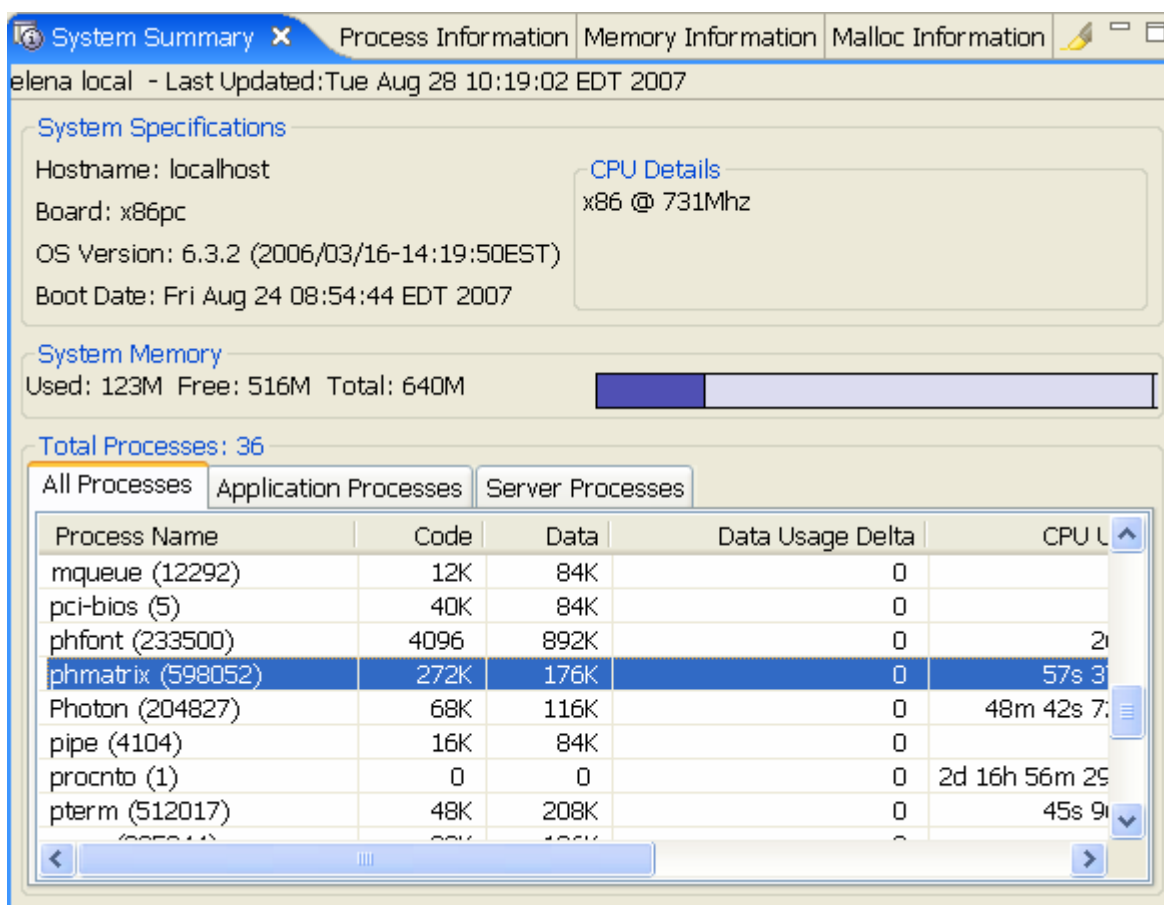
## Procedure 1: Inspecting process memory distribution

You can use the **System Information** view from QNX Momentics IDE 4 to inspect the distribution and overall memory occupation for the current process.

**Note:** The **IDE** must be currently running, you must have created a target project, and your target host must be connected.

**To inspect the process memory distribution:**

1. Run the process that you want to inspect on the target.
2. Switch to the **System Information** perspective.
3. In the **Target Navigator** view, select the target on which your process is running.
4. Switch to the **System Summary** view. In this view, you can obtain an overview of the process memory.
5. On the **All Processes** tab, select a process of interest.



The screenshot displays the 'System Summary' window in QNX Momentics IDE 4. The window has tabs for 'System Summary', 'Process Information', 'Memory Information', and 'Malloc Information'. The 'System Summary' tab is active, showing system specifications, CPU details, system memory usage, and a list of processes.

**System Specifications:**

- Hostname: localhost
- Board: x86pc
- OS Version: 6.3.2 (2006/03/16-14:19:50EST)
- Boot Date: Fri Aug 24 08:54:44 EDT 2007

**CPU Details:**

- x86 @ 731Mhz

**System Memory:**

- Used: 123M Free: 516M Total: 640M

**Total Processes: 36**

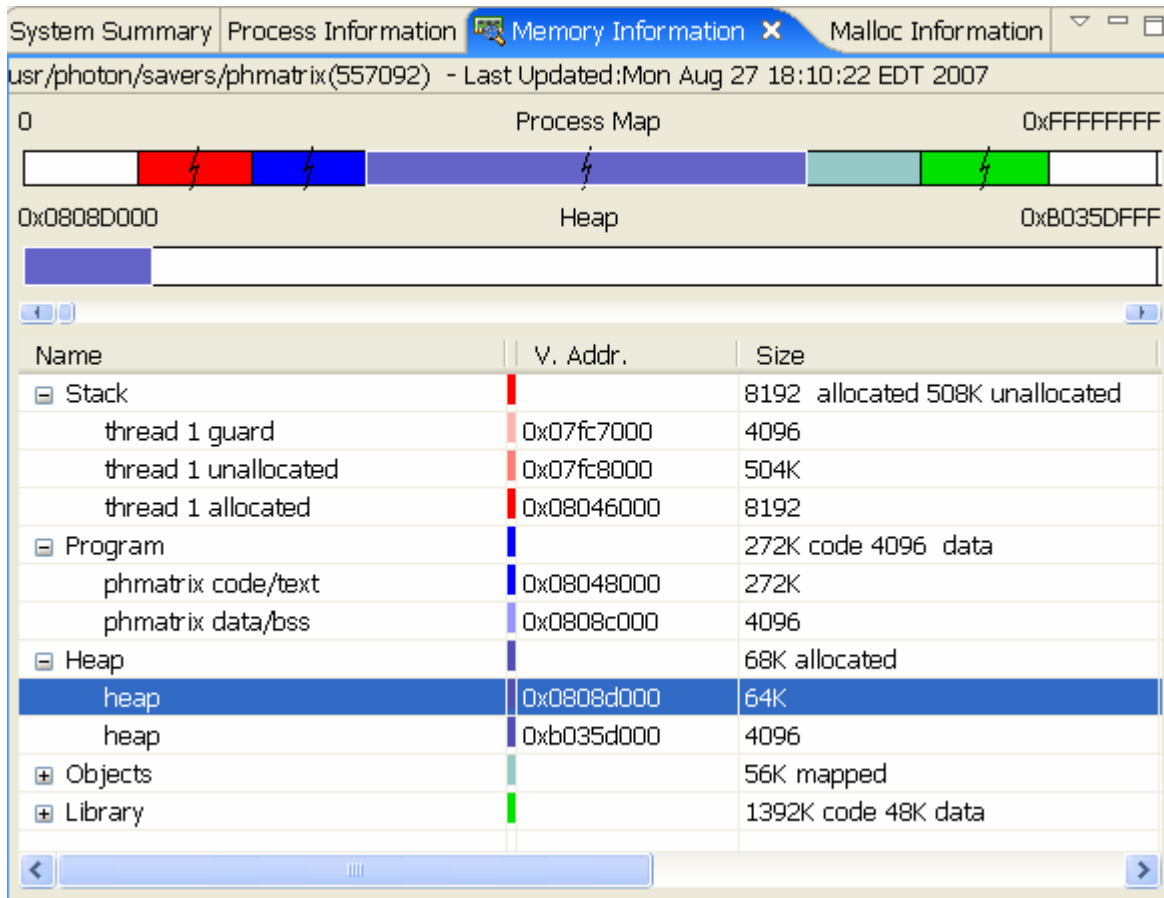
The 'All Processes' tab is selected, showing a table of processes. The 'phmatrix (598052)' process is highlighted.

Process Name	Code	Data	Data Usage Delta	CPU U
mqueue (12292)	12K	84K	0	
pci-bios (5)	40K	84K	0	
phfont (233500)	4096	892K	0	2i
<b>phmatrix (598052)</b>	<b>272K</b>	<b>176K</b>	<b>0</b>	<b>57s 3</b>
Photon (204827)	68K	116K	0	48m 42s 7
pipe (4104)	16K	84K	0	
procnto (1)	0	0	0	2d 16h 56m 25
pterm (512017)	48K	208K	0	45s 9i

From this illustration, you can see how much physical memory the selected process occupies; in this example, it is 272K of Code, and 176K of Data.

6. Now, switch to the **Memory Information** view.
7. In the Target Navigator, expand your target and select the same process you selected earlier.

8. You can see a detailed map of the virtual memory for the process.



Based on the memory distribution information in the preceding example, you can determine if it is ideal to allocate time to optimize the heap memory, or you might want to consider to optimize something else, such as the stack or static memory.

## Performance of heap allocations

Heap memory profiling can be performed to achieve two goals: performance improvements (because heap memory allocation/deallocation is one of the most expensive ways of obtaining memory), and heap memory optimization. The **QNX Momentics Memory Analysis** tool can assist you with both of these goals.

**Note:** We assume that your application is currently running without memory errors. Using the IDE tools to find memory errors is out of scope of this article.

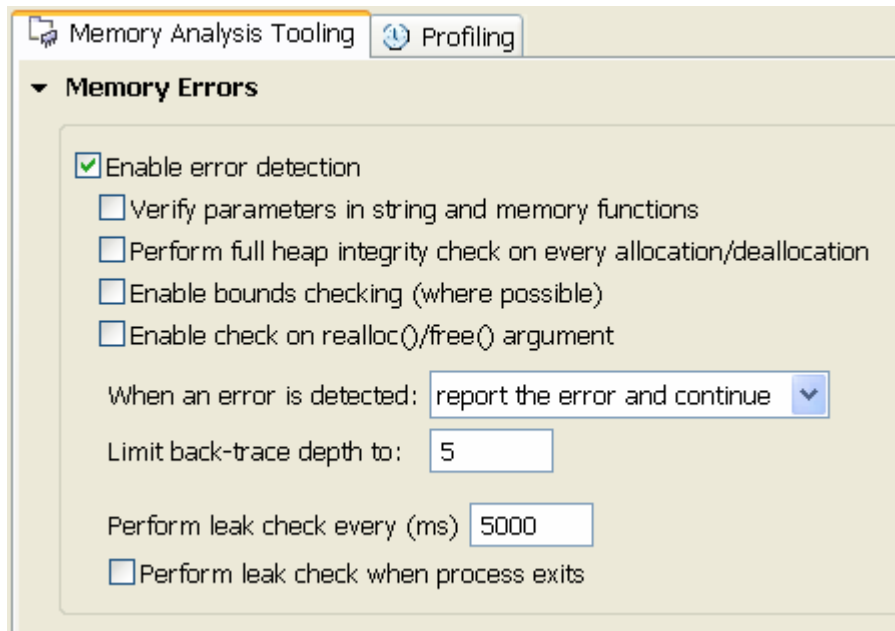
### Procedure 2: Preparing a memory profiling session

To prepare for a memory profiling session:

1. First, you will need to compile the binary with debug options. This configuration is required in order to link the results to source code.
2. Create a launch configuration to run your application on the target system.
3. In the **Launch Configuration** dialog, select the **Tools** tab

4. Click **Add Tool**, to enable the **Memory Analysis Tooling** option, and click **Ok**.
5. Expand the **Memory Errors** folder and disable (un-check) all items in the list, except for **Perform leak check when process exits**.
6. **Optional:** If your process never exits, edit the **Perform check every (ms)** option, and specify an interval in milliseconds. This value will be used to periodically perform a verification for memory leaks.

**Note:** It is sufficient to check only once each time you run the application because the leaks would be duplicated, and the leak detection process itself takes a significant amount of time to complete.



The screenshot shows the 'Memory Analysis Tooling' dialog box with the 'Profiling' tab selected. The 'Memory Errors' section is expanded, showing a list of checkboxes for error detection. The 'Enable error detection' checkbox is checked. Below it, four checkboxes are unchecked: 'Verify parameters in string and memory functions', 'Perform full heap integrity check on every allocation/deallocation', 'Enable bounds checking (where possible)', and 'Enable check on realloc()/free() argument'. A dropdown menu for 'When an error is detected:' is set to 'report the error and continue'. A text box for 'Limit back-trace depth to:' contains the value '5'. A text box for 'Perform leak check every (ms)' contains the value '5000'. The 'Perform leak check when process exits' checkbox is unchecked.

Memory Analysis Tooling Profiling

▼ **Memory Errors**

- ☒ Enable error detection
  - ☐ Verify parameters in string and memory functions
  - ☐ Perform full heap integrity check on every allocation/deallocation
  - ☐ Enable bounds checking (where possible)
  - ☐ Enable check on realloc()/free() argument

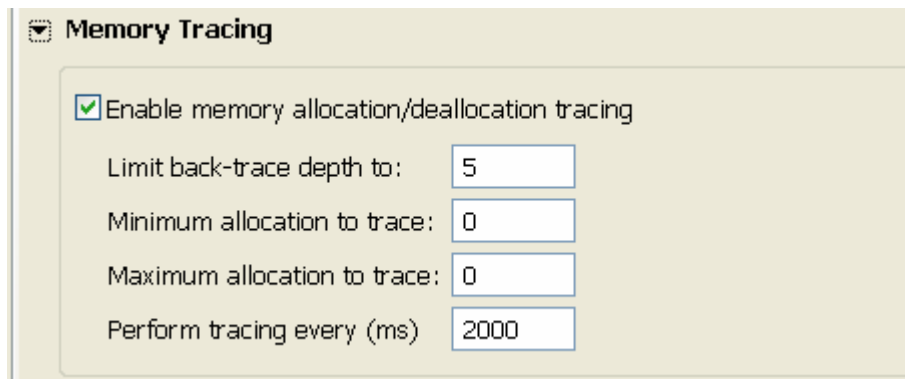
When an error is detected: report the error and continue ▼

Limit back-trace depth to: 5

Perform leak check every (ms) 5000

☐ Perform leak check when process exits

7. Expand the **Memory Tracing** folder. Ensure that you enable the **Enable memory allocation/deallocation tracing** option.



The screenshot shows the 'Memory Tracing' section of the dialog box. The 'Enable memory allocation/deallocation tracing' checkbox is checked. Below it, four text boxes are visible: 'Limit back-trace depth to:' with value '5', 'Minimum allocation to trace:' with value '0', 'Maximum allocation to trace:' with value '0', and 'Perform tracing every (ms)' with value '2000'.

☑ **Memory Tracing**

- ☒ Enable memory allocation/deallocation tracing

Limit back-trace depth to: 5

Minimum allocation to trace: 0

Maximum allocation to trace: 0

Perform tracing every (ms) 2000

- Expand the **Memory Snapshots** tab. Ensure that you enable the **Memory Snapshots** option, and type an interval for the snapshots for your application (i.e., 10 to 20 snapshots during the entire application execution).

▼ **Memory Snapshots**

☒ Memory Snapshots

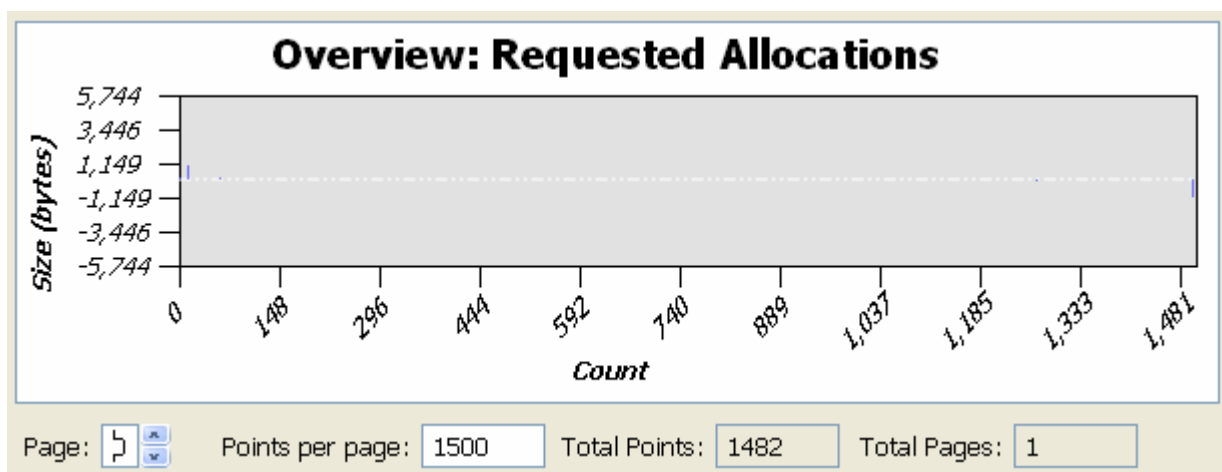
Perform snapshot every (ms)

Bins counters (comma separated) e.g. 8,16,32,1024

- If you use custom shared libraries, expand the **Library search paths** tab, and specify information so that the tool can also read symbol information from the libraries.
- Enable the **Switch to this tool's perspective on launch** option at the bottom of the page.
- Now, you can launch the application. The IDE switches to the **Memory Analysis** perspective. You will see a new session display in the **Session View**. Let the application run for a desired amount of time (you may perform a testing scenario), and then stop it (either it should terminate itself or you can stop it from IDE).
- Now, the **Memory Analysis** session will be ready, and we can begin to inspect the results.

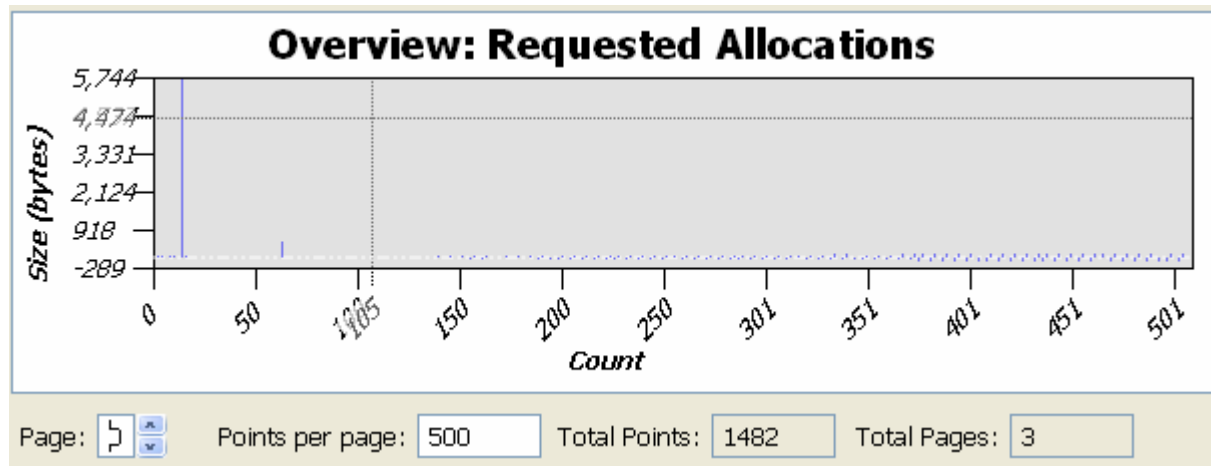
## Analyzing allocation patterns

Once we've prepared a memory profiling session, to begin our analysis we need to open the **Memory Analysis** session viewer by double-clicking on a session. First thing you'll see is the **Allocations** page with the **Overview: Requested Allocations** chart. Let's take a closer look at this chart.

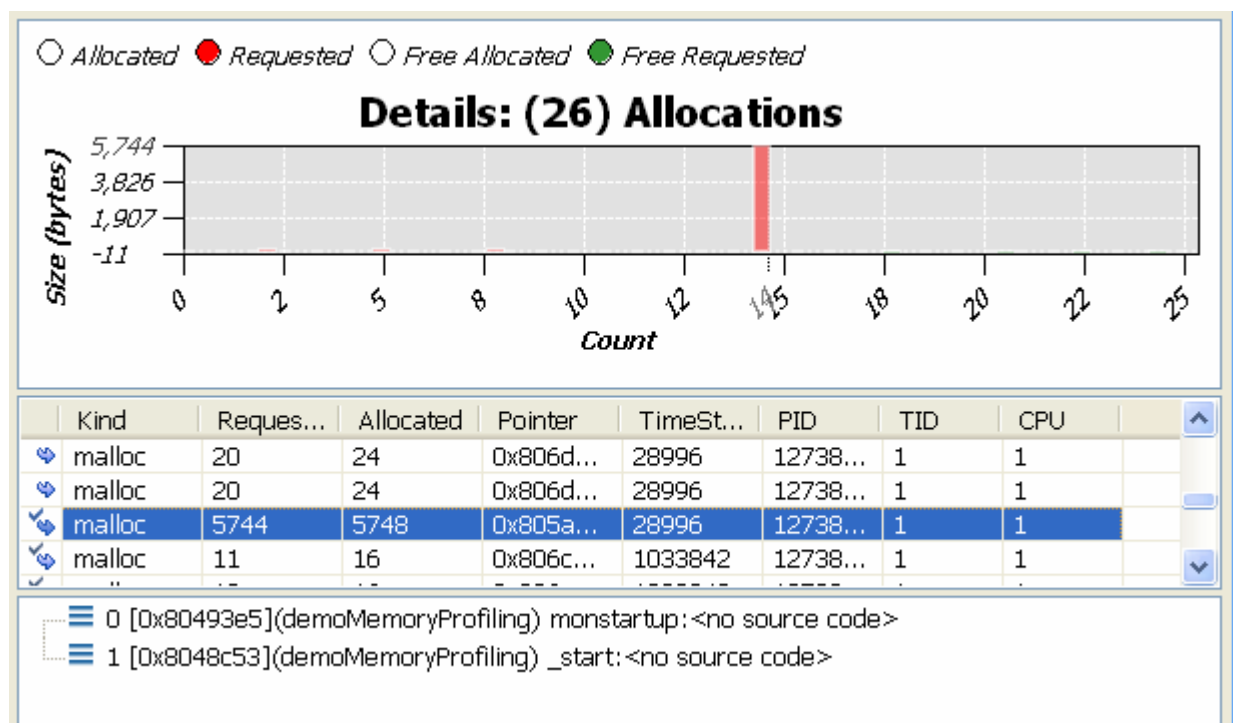


At first glance, it does not look like a very useful chart, does it? Let's see what we can do about it. This chart shows memory allocation and deallocation events generated by the `malloc` and `free` functions and their derivatives. The **X-axis** represents the event number (which can be changed to be the timestamp), and the **Y-axis** represents the size (in bytes) of the allocation (if a positive value), or deallocation (if a negative value). Let's take a closer look at the bottom portion of the chart. The **Page** field shows the scrollable page number, the **Total Points** field shows how many recorded events there are, the **Points per page** field shows how many events can fit onto this page, and the **Total Pages** field shows how many chart pages there are in total.

For our example, there are 1482 events, and all of them would not likely fit on this single chart. There are several choices we have available to us. First, we can attempt to reduce the value in the **Points per page** field to 500, for example. Now, the graphical representation is better, but it is not very useful.



If you look at Y-axis, you can notice some big allocations at the beginning. To see this area more closely, select this region with the mouse, and the chart and table at the top change to populate with the data from the selected region. Now, if we locate our big allocation and check its stack trace, we can see that this allocation belongs to the function called `monstartup`, which is not part of our own code, meaning that it cannot be optimized and we should probably exclude it from the events of interest.



To exclude this function, we can use a filter. Right-click on the **Overview** chart's canvas, select **Filters...** from the menu, and the **Filter** dialog will appear. Type **1000** in the **To allocation size** field.

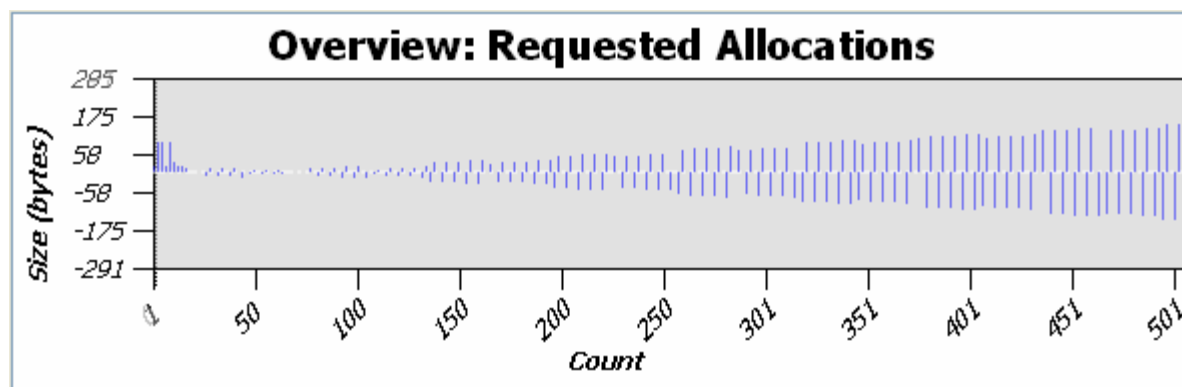
From allocation size:

0

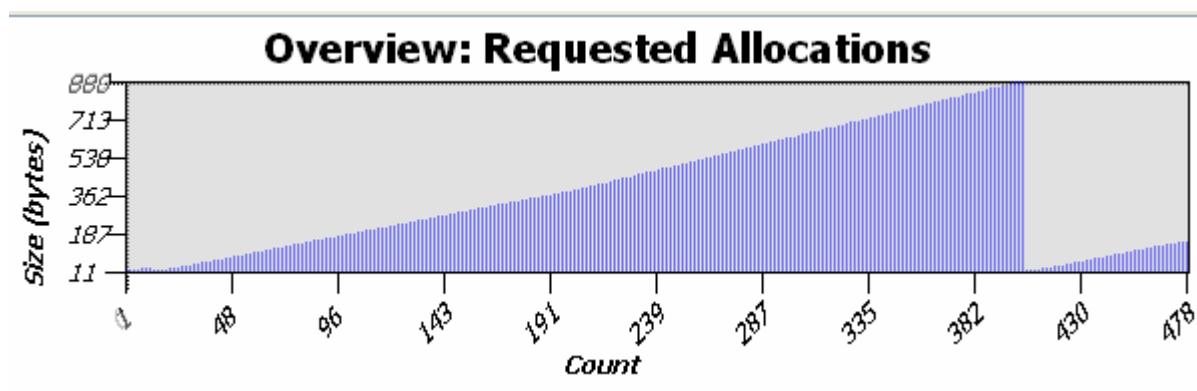
To allocation size:

1000

The overview will look like this:



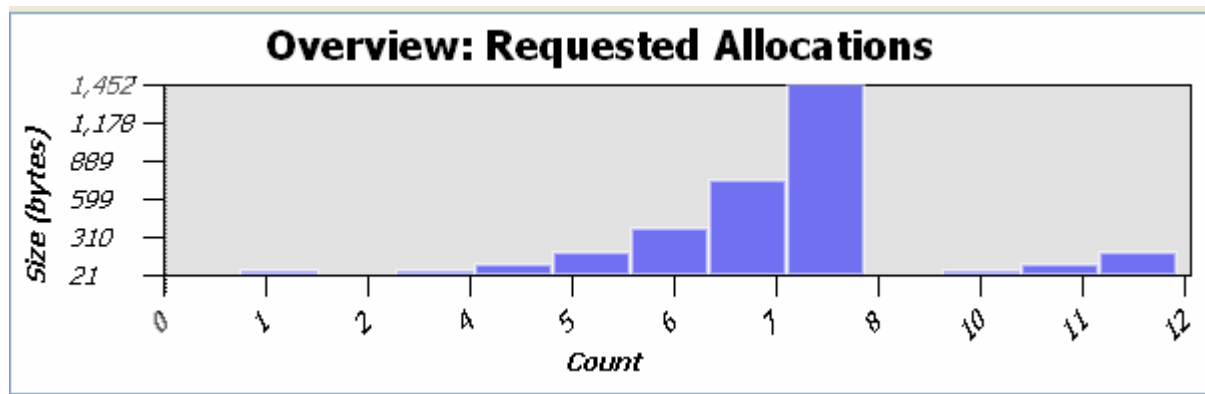
From the filtered view, we can see a pattern: the allocation is followed by a deallocation, and the size of the allocations grows over time. Typically, this growth is the result of the `realloc` pattern. To confirm our suspicion, return to the **Filters...** menu option, and disable (un-check) all of the allocation functions, except for the **realloc** option. Now, we can clearly see that the growth occurs with a very small increment.



Next, select some region of the overview chart and explore the event table. You will notice events with the same stack trace. This is one `realloc` call with a bad (too small) increment. This would be the pattern for a shortsighted `realloc`.

```
29 {
30     int n = pstr->size;
31     int l = pstr->len;
32     int lx = strlen(x);
33     if (l+lx>=n) {
34         int size = l+lx+1; // realloc size
35         str = (char *)xrealloc(str,size);
36         pstr->size=size;
37         pstr->str=str;
38     }
39 }
```

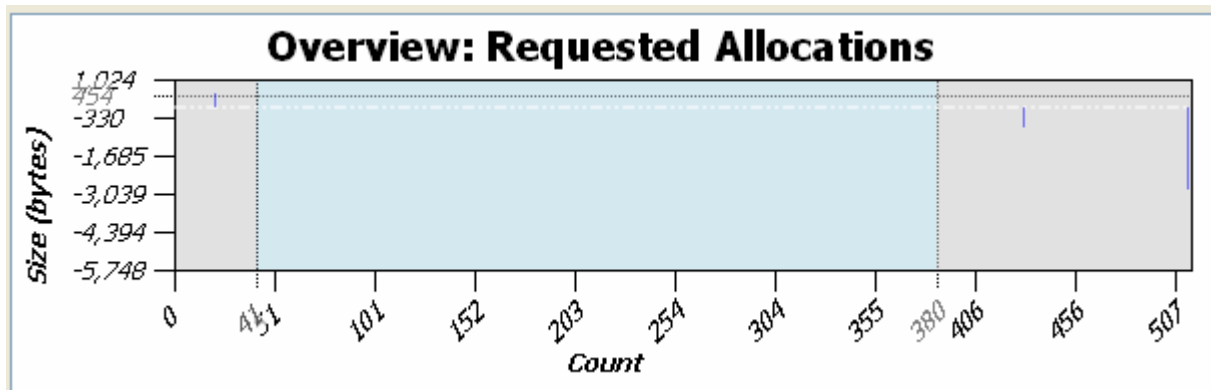
Notice that the string in the example was re-allocated approximately 400 times (from 11 bytes to 889 bytes). Based on that information, we can optimize this particular call (for performance) by either adding some constant overhead to each `realloc` call, or double allocate the size. In this particular example, we'll double allocate the size, and then re-compile and re-run the application. Open the editor and filter all but the `realloc` events:



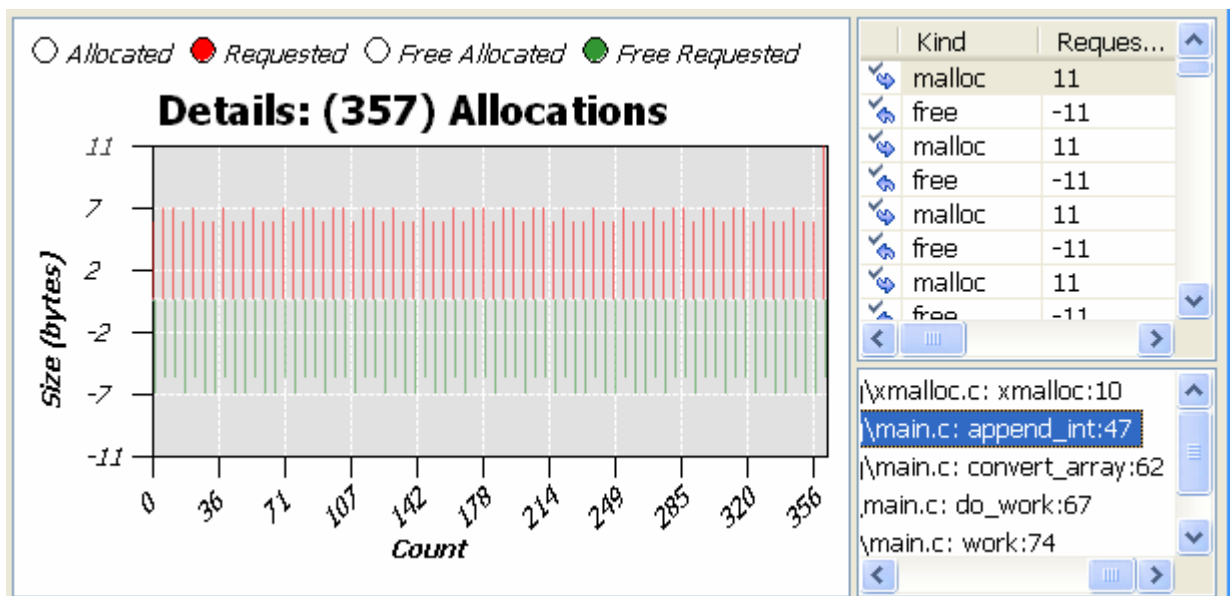
Now, we see only 12 `realloc` events instead of the original 400, which would significantly improve the performance; however, the maximum allocated size is 1452 bytes, which is 600 bytes in excess of what we require. We can adjust the `realloc` code to better tune it for a typical application run. Normally, you should make `realloc` sizes similar to the allocator block sizes. (This topic is discussed in more detail in the next section called, "Optimizing heap memory".)



Now, let's return to check other events. In the Filters menu, enable all functions, except for `realloc` (ensure that `realloc` remains un-checked). Select a region in the overview:



In the **Details** chart, the **alloc/free** events have the same size. This is the typical pattern for a short-lived object.



Navigate to the source code from the stack trace view (double-click on a row for the stack trace):

```

46 int append_int(String* pstr, int i) {
47     char * str = (char *)xmalloc(11);
48     if (str==NULL) return 1;
49
50     // integer to string conversion
51     itoa(i, str, 10);
52     append_str(pstr, str);
53     free(str);
54     return 0;
55 }

```

This code has an object where it allocates 11 bytes, and then it is freed at the end of the function. This is a good candidate to put a value on the stack. However, if the object has a variable size, and comes from the user, using stack buffers should be done carefully. As a compromise between performance and security, a size check can be performed, and if the length of the object is less than the buffer size, it is safe to use the stack buffer; otherwise, if it is more than the buffer size, the heap can be allocated. The buffer size can be chosen based on the average size of allocated objects for this particular stack trace (refer to Procedure 6 below).

Short-sighted `realloc` functions and short-lived objects are memory allocation patterns which can improve performance of the application, but not the memory usage.

## Optimizing Heap Memory

You can use the following techniques to optimize memory usage:

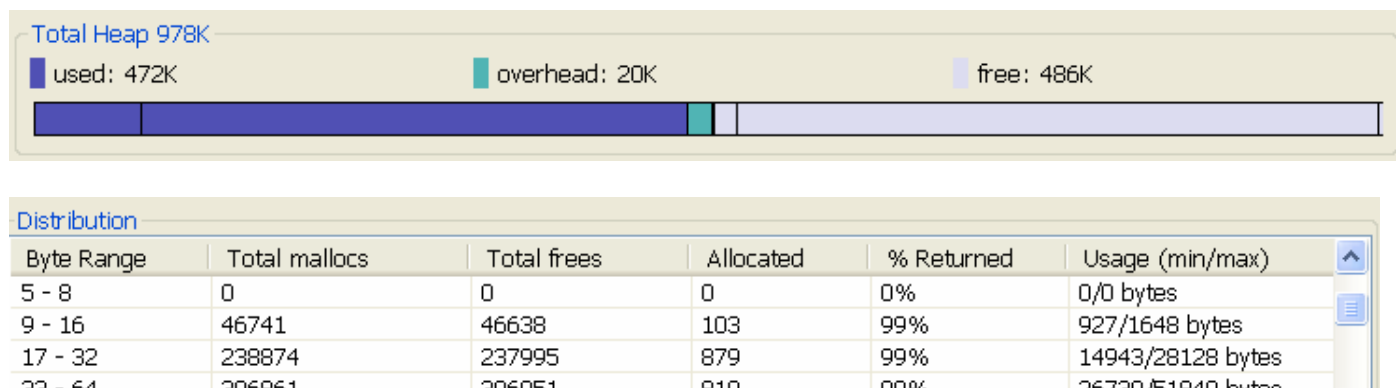
- Eliminate memory leaks
- Shorten the life cycle of heap objects
- Reduce the overhead of allocated objects
- Tune the allocator

## Memory Leaks

A memory leak is a portion of heap memory that was allocated but not freed, and the reference to that area of memory cannot be used by the application any longer. Typically, the elimination of a memory leak is critical for applications that run continuously because even a single byte leak can crash a mission critical application that runs over time.

### Procedure 3: Quick check for Memory Leaks

To test an application for memory leaks, run the application and then compare memory usage at specific times. You can use the **Malloc Information** view for this by selecting a process, and then watching the **Allocated** column (which is **Outstanding allocations** in memory) and observe the number in this column to see if it increases (see the growth in the chart below). If you notice a trend where it increases over time, then the process is not returning some of the allocated memory.



To know exactly what's occurring, use the **Memory Analysis** tool.

Memory leaks can be apparent or hidden. Apparent memory leaks can be found by the tool automatically. A memory leak is "apparent" if the binary address of that heap block (marked as allocated) is not stored in any of the process memory and current CPU registers any longer.

There are three ways of finding memory leaks using the QNX Momentics Memory Analysis tool. To perform an automatic leak check, use the following two options from the Memory Analysis Tooling launch configuration: **Perform leak check when process exits** and **Perform leaks check every (ms)** with your desired interval. If these options are enabled, the Memory Analysis tool automatically checks for memory leaks in the currently running program.

Perform leak check every (ms)

☐ Perform leak check when process exits

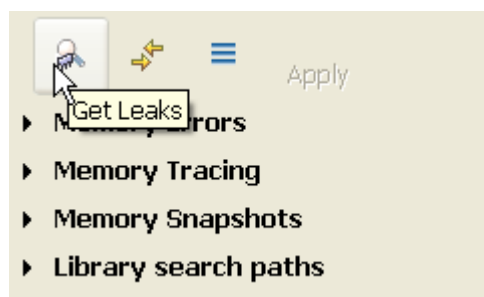
The following illustration shows a list of memory leaks that memory Analysis tool identified:

	Severity	Pointer	Timestamp	PID	TID	Operation	
	LEAK	0x8052290	20254900	1568810	1	malloc	
	LEAK	0x8052478	20254900	1568810	1	malloc	
	LEAK	0x8052590	20254900	1568810	1	malloc	

0	[0x8048868](testMallocPerformance)	c:\Develop\runtime-test\testMallocPerformance\main.c: allocateTree:90
1	[0x80489df](testMallocPerformance)	c:\Develop\runtime-test\testMallocPerformance\main.c: malloc_test:123
2	[0x8048b25](testMallocPerformance)	c:\Develop\runtime-test\testMallocPerformance\main.c: main:136
3	[0x80485af](testMallocPerformance)	_start: <no source code>

The third and final method to find a memory leak is to use the **Get Leaks** button on the **Settings** page of Memory Analysis Session viewer while application runs (which can also be used the debugger).



## Procedure 4: Enabling memory leaks detection

The following procedure (for example, in a continuously running application) will enable memory leak detection at any particular point in program execution:

1. Find a location in the code where you want to check for memory leaks, and insert a breakpoint.
2. Launch the application in Debug mode with **Memory Analysis Tooling** enabled (for instructions, see Procedure 2).
3. Switch to the **Memory Analysis** perspective.
4. Open the **Debug** view so it is available in the current perspective.

5. When the application encounters the breakpoint you specified, open the Memory Analysis session from the **Session View** (by double-clicking) and switch to the **Setting** page for the Session Viewer.
6. Click the **Get Leaks** button. Before you resume the process, no new data will be available in the Session Viewer because the memory analysis thread and application threads are stopped while the process is suspended by the debugger.
7. click the **Resume** button in the **Debug** view to resume the process' threads.
8. If leaks did not appear on the **Errors** tab of the Session Viewer, either there were no leaks, or the time given to the control thread (a special memory analysis thread that processes dynamic requests) to collect the leaks was not long enough to perform the command (it was suspended before the operation completed).
9. Now, switch to the **Errors** page of the viewer, and you can review information about collected memory leaks.

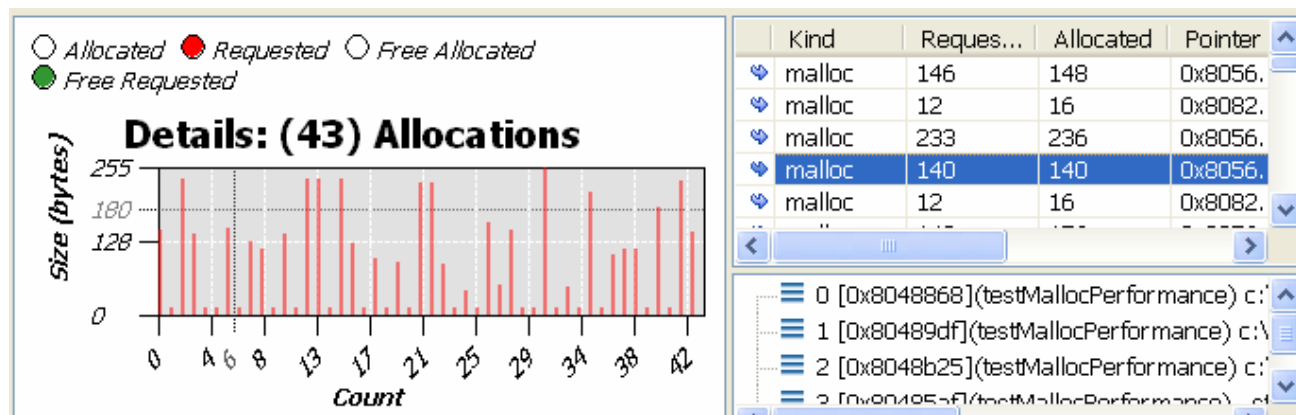
Besides apparent memory leaks, an application can have other types of leaks that the memory Analysis tool cannot detect. These leaks include objects with cyclic references, accidental point matches and left-over heap references (which can be converted to apparent leaks by nullifying objects that refer to the heap. If you can continue to see the heap grow after eliminating apparent leaks, you should manually inspect some of the allocations. You can do this review after the program terminates (completes), or you can stop the program at any time using the debugger, and inspect the current heap state.

## Procedure 5: Manually inspecting outstanding allocations

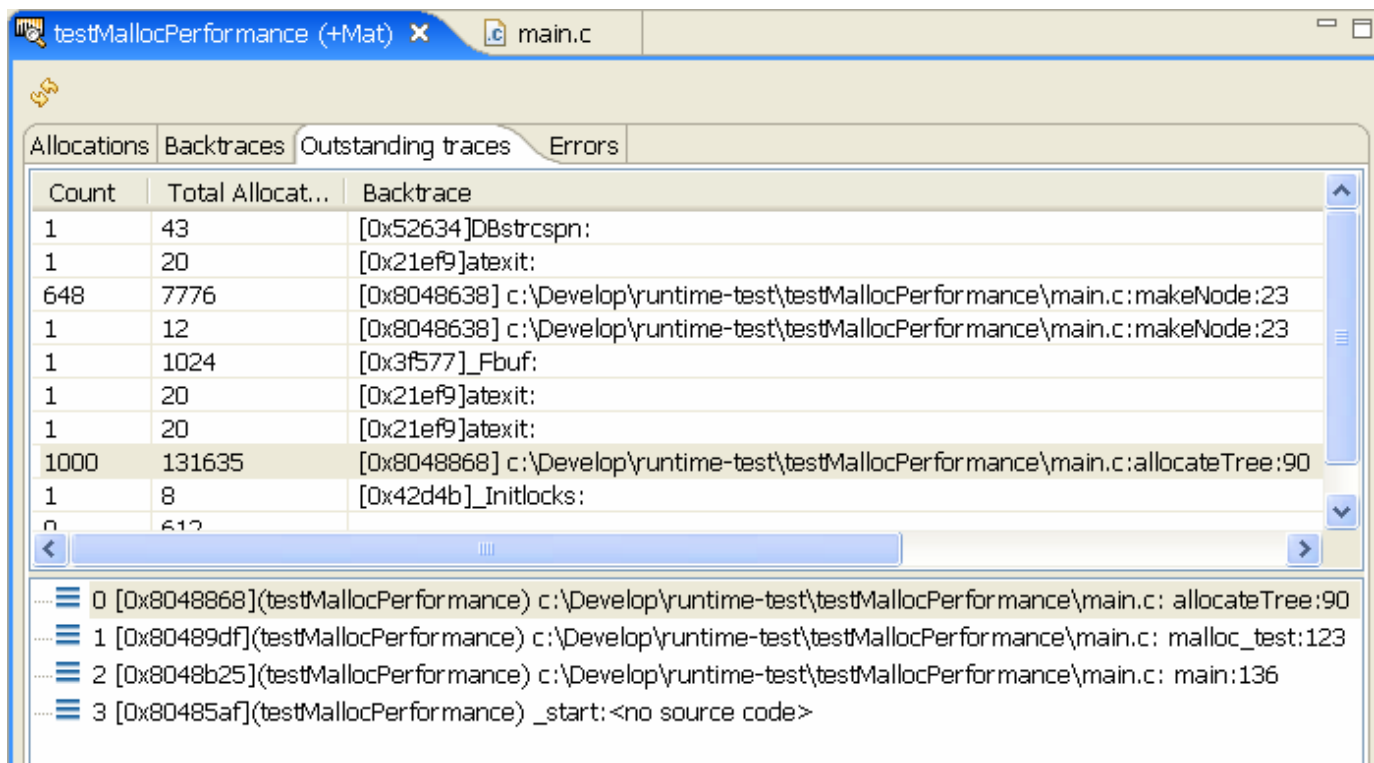
To manually inspect outstanding allocations:

1. Open the **Allocations** page of the Session Viewer.
2. In the **Overview** chart, select the **Filters...** option from the menu.
3. Enable the **Hide matching allocation/deallocation pair** option and click **Ok**.
4. Review the results (only those allocations which continue to be in memory or were in memory at the moment of the exit).
5. Select the allocations using the mouse. The **Details** view and table become populated with the data.
6. Select one allocation from the table. The **Trace** view becomes populated with the current stack trace for the selected event.

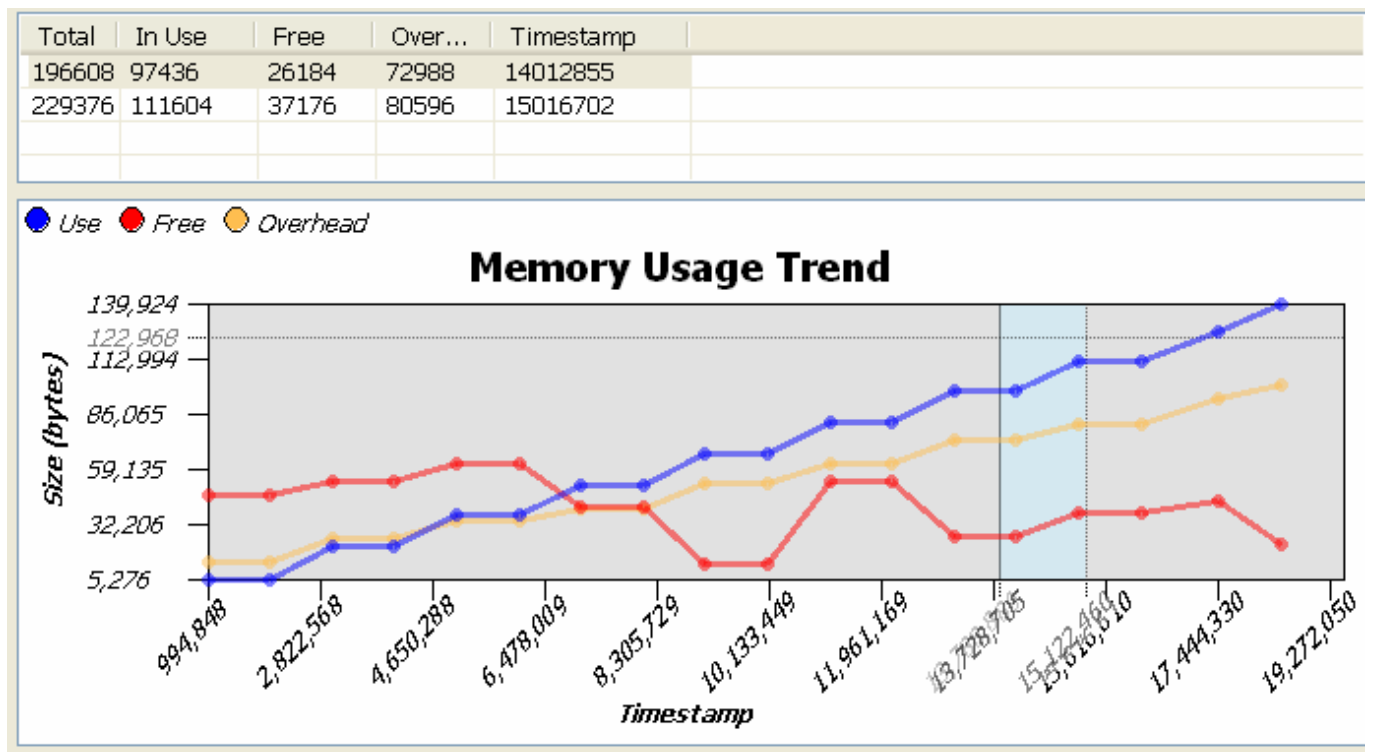
You can inspect the stack trace of the allocation (using source code navigation) and determine whether it is leak.



The same data that is grouped by backtraces is available from the **Statistics** page on the **Outstanding traces** tab.

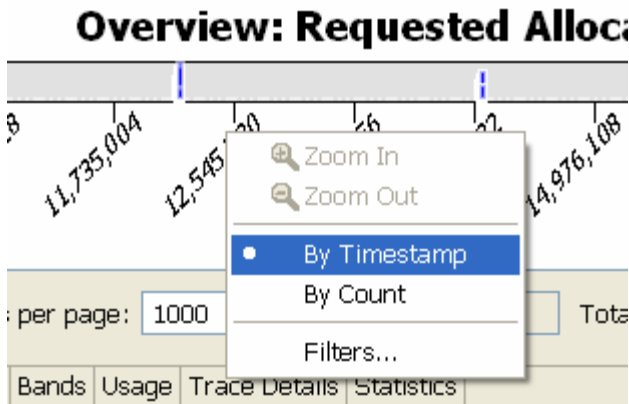


Alternatively, you can inspect memory allocations that occurred during a particular time frame of interest by opening the **Usage** page, which will be populated with snapshots of memory usage (if you enabled this option).

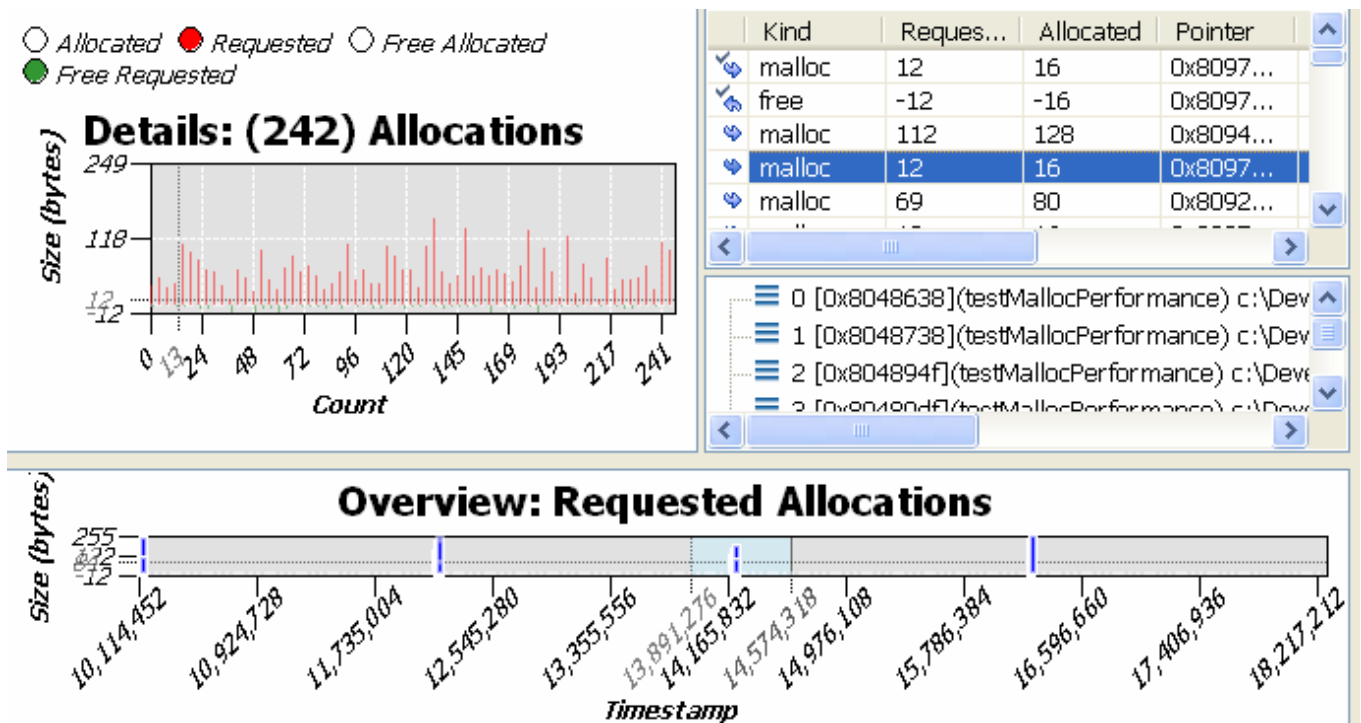


Select the region you are interested in and the statistics for the memory usage will populate the table above.

Select the **Allocations** page and switch the X-axis mode to be **timestamp** (changes to the timestamp label):



Select an area of interest. You would see allocations and de-allocations that occurred during this period in the **Details** chart and event table. Review the event table results. All `malloc` calls that do not have a paired `free` would not have icon in the left column with a checkmark beside it.



## Object life cycle

The other optimization technique is to shorten the heap objects' life cycle. This technique lets the allocator reclaim memory faster, and allows it to be immediately used for new heap objects, which, over time, reduces the maximum amount of memory required.

Always attempt to free objects in the same function as they are allocated, unless it is an allocation function. An allocation function is a function that returns or stores a heap object that would be used after this function exits. A good pattern of local memory allocation will look like this:

```
...
p=(type *)malloc(sizeof(type));
do_something(p);
free(p);
p=NULL;
do_something_else();
...
```

As a result, after the pointer is used, you free it and then you nullify it so that the free block cannot be referred to, and then you do something else, making sure that during this other activity, the memory is already freed. In addition, try to avoid creating aliases for heap variables because it usually makes code less readable, prone to errors, and difficult to tools to analyze.

Use Procedure 5, described above, to find currently allocated memory, and then inspect it so see if you can shorten an objects' cycle.

## Allocation overhead

Another large source of memory usage occurs from the following types of allocation overhead:

- User overhead — The actual data occupies less memory when requested by the user
- Padding overhead — The fields in a structure are arranged in a way that the **sizeof** of a structure is larger than the sum of the **sizeof** of all of its fields.
- Heap fragmentation — The application takes more memory than it needs, because it requires contiguous memory blocks, which are bigger than chunks that allocator has
- Block overhead — The allocator actually takes a larger portion of memory than required for each block
- Free blocks — All free blocks continue to be mapped to physical memory

**User overhead** usually comes from predictive allocations (usually by `realloc`), which allocate more memory than needed. You can either tune it by estimating the average data size, or - if your data model allows it - after the growth of data stops, you can truncate the memory to fit into the actual size of the object.

### Procedure 6: Estimating the average allocation size

To estimate the average allocation size for a particular function call, find the backtrace of a call on the **Backtraces** tab of the **Statistics** page of the Session Viewer. The **Count** column represents the number of allocations for a particular stack trace, and the **Total Allocated** column shows the total size of the allocated memory. To calculate an average, divide the **Total Allocated** value by the **Count** value.

The screenshot displays a memory profiler interface with two main sections. The top section shows a table of allocations, and the bottom section shows a backtrace for a specific allocation.

Count	Total Allocated	Backtrace
1	1024	[0xb033f577]
1	-29	[0x8048eef] c:\Develop\runtime-test\demoMemoryProfiling\main.c:do_work:68
200	2200	[0x8048dce] c:\Develop\runtime-test\demoMemoryProfiling\main.c:append_int:45
1	10	[0x8048cef] c:\Develop\runtime-test\demoMemoryProfiling\main.c:append_str:21
200	-2200	[0x8048e18] c:\Develop\runtime-test\demoMemoryProfiling\main.c:append_int:51

The backtrace for the selected allocation (0x8048dce) is as follows:

- 0 [0x8048dce](demoMemoryProfiling) c:\Develop\runtime-test\demoMemoryProfiling\main.c: append\_int:45
- 1 [0x8048e89](demoMemoryProfiling) c:\Develop\runtime-test\demoMemoryProfiling\main.c: convert\_array:
- 2 [0x8048ebb](demoMemoryProfiling) c:\Develop\runtime-test\demoMemoryProfiling\main.c: do\_work:65
- 3 [0x8048f1d](demoMemoryProfiling) c:\Develop\runtime-test\demoMemoryProfiling\main.c: work:72
- 4 [0x8048f98](demoMemoryProfiling) c:\Develop\runtime-test\demoMemoryProfiling\main.c: main:86

The bottom section shows the source code for `main.c` with the following content:

```

42     return 0;
43 }
44 int append_int(String* pstr, int i) {
45     char * str = (char *)malloc(11);
46     if (str==NULL) return 1;
47
48     // integer to string conversion

```

**Padding overhead** affects the **struct** type on processors with alignment restrictions. The fields in a structure are arranged in a way that the `sizeof` of a structure is larger than the sum of the `sizeof` of all of its fields. You can save some space by re-arranging the fields of the structure. Usually, it is better to group fields of the same type together. You can measure the result by writing a **sizeof** test. Typically, it is worth performing this task if the resulting **sizeof** matches with the allocator block size (see below).

**Heap fragmentation** occurs when a process uses a lot of heap allocation/deallocation of different sizes. When this happens, the allocator divides large blocks of memory into smaller ones, which later cannot be used for bigger blocks because the address space is not contiguous. In this case, the process will allocate another physical page even if it looks like it has enough free memory. The QNX memory allocator is a "bands" allocator, which already solves most of this problem by allocating blocks of memory of constant sizes of 16, 24, 32, 48, 64, 80, 96 and 128 bytes. Having only a limited number of possible sizes lets the allocator choose the free block faster, and keeps the fragmentation to a minimum. If a block is more than 128 bytes, it is allocated in a general heap list, which means a slower allocation and more fragmentation. You can inspect the heap fragmentation by reviewing the **Bins** or **Bands** graphs. An indication of an unhealthy fragmentation occurs when there is growth of free blocks of a smaller size over a period of time.



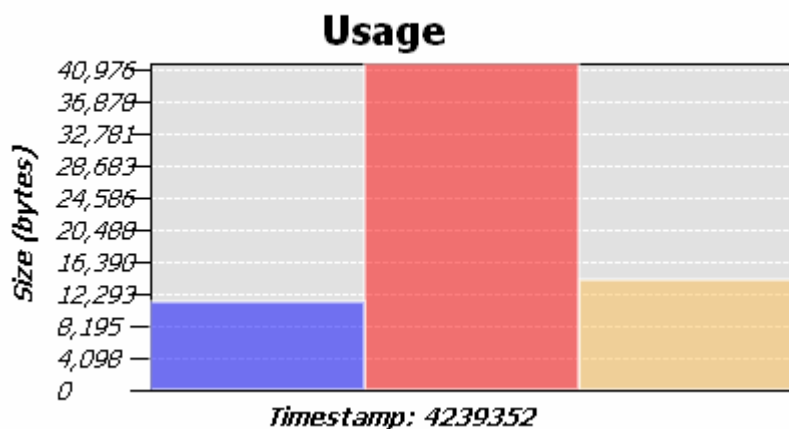
**Block overhead** is a side effect of combating heap fragmentation. Block overhead occurs when there is extra space in the heap block; it is the difference between the user requested allocation size and actual block size. Block overhead is available to inspect using the Memory Analysis tool:

	Kind	Requested	Allocated	Pointer	Time
	free	-11	-16	0x806c188	10338
	malloc	11	16	0x806c188	10338
	free	-11	-16	0x806c188	10338
	realloc...	-43	-48	0x806d430	10338

In the allocation table, you can see the size of the requested block (11) and the actual size allocated (16).

You can also estimate the overall impact of the block overhead by switching to the **Usage** page:

● Use ● Free ● Overhead



You can see in this example that current overhead is larger than the actual memory currently in use.

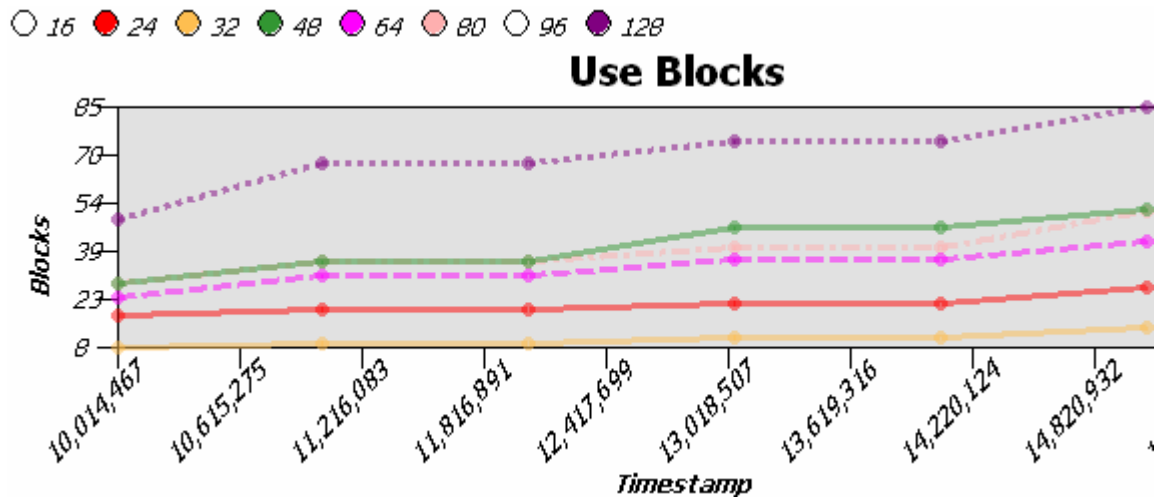
Some techniques to avoid block overhead are:

- You should consider allocator band numbers, when choosing allocation size, especially for predictive `realloc`. This is the algorithm that can give you next highest power or two for a given number  $m$  if it is less than 128, or a 128 divider if it is more than 128:

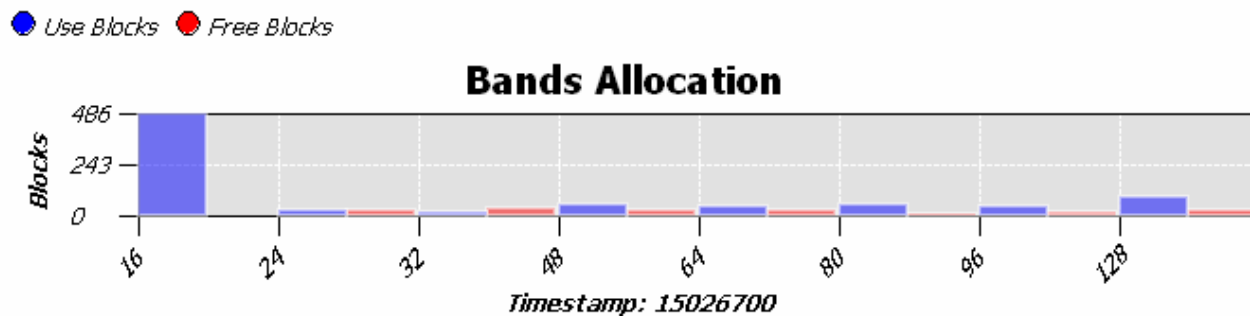
```
int n;
if (m > 256) {
    n = ((m + 127) >> 7) << 7;
} else {
    n = m - 1;
    n = n | (n >> 1);
    n = n | (n >> 2);
    n = n | (n >> 4);
    n = n + 1;
}
```

It will generate the following size sequence: 1,2,4,8,16,32,64,128,256,384,512,640, and so on.

- You can attempt to optimize data structures to align with values of the allocator blocks (unless they are in an array). For example, if you have a linked list in memory, and a data structure does not fit within 128 bytes, you should consider dividing it into smaller chunks (which may require an additional allocation call), but it will improve both performance (since band allocation is generally faster), and memory usage (since there is no need to worry about fragmentation). You can run the program with **Memory Analysis Tooling** enabled again (using the same options), and compare the **Usage** chart to see if you achieved the desired results. You can observe how memory objects were distributed per block range using **Bands** page:



This chart shows, for example, that at the end there were 85 used blocks of 128 bytes in a block list. You also can see the number of free blocks by selecting a time range.



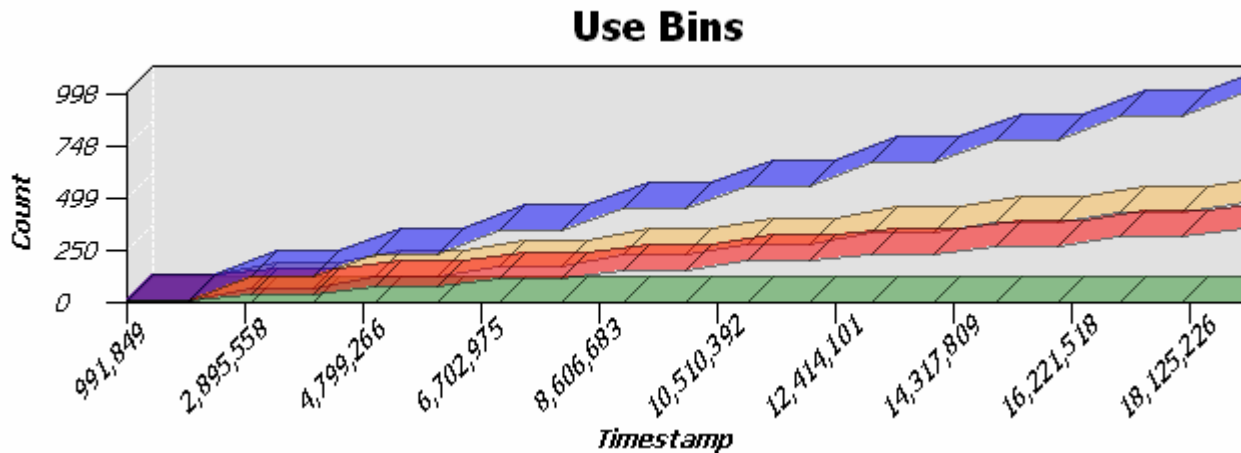
Timestamp	Size	Use	Free	
15026700	128	85	20	
15026700	96	40	10	
15026700	80	51	5	

And finally, **Free Blocks** overhead. When you free memory using the `free` function, memory is returned to the process pool, but it does not mean that the process will free it. When the process allocates pages of physical memory, they are almost never returned. However, a page can be deallocated when the ratio of used pages reaches the low water mark. Even in this case, a virtual page can be freed only if it consists entirely of free blocks.

# Tuning the allocator

Occasionally, application driven data structures have a specific size, and memory usage can be greatly improved by customizing block sizes. In this case, you either have to write your own allocator, or contact QNX to obtain a customizable memory allocator. To estimate the benefits of a custom block size, you can use the **Bin** page. First, enter the bin size in the Launch Configuration of the Memory Analysis tool, run the application, and then open the **Bins** page to explore the results. The resulting graph shows the distribution of the heap object per imaginary blocks, based on the sizes that you selected.

● 32 ● 128 ● 1024 ● 10000



## Optimizing Static and Stack Memory

In the previous section, we explained tool-assisted techniques for optimizing heap memory, and now we will describe some tips for optimizing static and stack memory:

### Code

In embedded systems, it is particularly important to optimize the size of a binary, not only because it takes RAM memory, but also because it uses expensive flash memory. Below are some tips you can use to optimize the size of an executable:

- Ensure that the binary is compiled without debug information when you measure it. Debug data is the largest contributor to the size of the executable, if it is enabled.
- Strip the binary to remove any remaining symbol information
- Remove any unused functions
- Find and eliminate code clones
- Try compiler performance optimization flags, such as `-O`, `-O2` (note that there is no guarantee that code would be smaller; it can actually be larger in some cases).
- Do not use the **char** type to perform **int** arithmetics, particularly when it is a local variable. Converting to **int** and back (code inserted by the compiler) affects performance and code size (particularly on ARM).
- Bit fields are also very expensive in arithmetics on all platforms; it is better to use bit arithmetics explicitly to avoid hidden costs of conversions.

## Data

- Inspect global arrays that significantly contribute to static memory consumption. In some cases, it may be better to use the heap, particularly when this object is not used through the entire process life cycle.
- Find and remove unused global variables
- Be aware of structure padding. Consider re-arranging fields to achieve smaller structure size.

## Stack

In some cases, it is worth the effort to optimize the stack, particularly when the application has some frequent picks of stack activity (meaning that a huge stack segment would be constantly mapped to physical memory). You can watch the **Memory Information** view for stack allocation and inspect code that uses stack heavily. This usually occurs in two cases: recursive calls (which should be avoided in embedded systems), and heavy usage of local variables (keeping arrays on the stack).

**Note:** Tasks of finding unused objects, structures that are not optimal, and code clones are not automated in the QNX Momentics IDE. You can search for static analysis tools with given keywords to find an appropriate tool for this task.

## Conclusions

This article introduces techniques and methods for performing memory profiling for embedded applications using QNX Momentics IDE tools. The non-intrusive QNX memory analysis tools can greatly simplify the labor-intensive job of memory profiling, which otherwise would have been done using a debugger, printf, or modifying application code.

For more information about QNX Momentics IDE 4, see <http://www.qnx.com>.