# New / Malloc Object-type profiling: unveiling dark matter in the heap

**Authors:** ssid@chromium.org **Last edit:** March 2016

**Parent doc:** Heap Profiling in MemoryInfra

## Problems

1. Dark matter: in normal (i.e. no --enable-heap-profiling) memory-infra dumps malloc dumper value is shown as large number without details.
2. In the near future MemoryInfra will have malloc heap profiling with pseudo stack traces - Tracking bug 580114. This gives us stack traces of only the categories enabled when tracing. The stack traces contains the a few important functions marked by TRACE_EVENT macros but sometimes they don't give enough context (not enough coverage).
3. The type info is very distributed and mainly has basic types like int and char. Who should we blame for this. How do we aggregate the types smarter?
4. Some information about the usage cannot be shown by adding such macros when they are done by external libraries. But, some of these can be covered by dump providers. We need a correlation between the sub-allocations of dump providers from malloc and types / stack trace info.

## Table of Contents

## Related docs

- Heap Profiling in MemoryInfra
- [tasak] Factor out Skia's memory from malloc
- Unified Malloc Shim Layer
- dskiba's CPP profiler
- dskiba's instrumenting std::string experiments

## Obtaining type-info

The basic ideas in the doc include:

1. Mark the components using the info stored at where the task is posted from. See FROM_HERE used in PostTask calls.
2. Group them using the thread names or category names from trace events.
3. Get types from scoped_ptr, scoped_refptr (CL).
4. Get types from std::allocator. Insert a custom implementation of std::allocator.
5. Group allocations using heap profiler at one or two level higher from malloc. Eg:sk_malloc, V8 New(), etc..
6. Add more TRACE_EVENT with new category "memory-profiling" which captures the functions of interest.

More detailed explanations below.

## Adding more TRACE_EVENT(S)

Build a call graph for malloc like this one. The idea is to try to get close to the native stack traces. Take 2 approaches:

Looking at graph from top-down. See the major functions where there are lots of branches and those which allocate a lot of memory. The points of divergence. Add TRACE_EVENT with new category “memory-profiling” which captures these functions.

1. cc::Display::DrawAndSwap, cc::LayerTreeHostImpl, UncompressEVWhitelist, ProfileManager::GetProfile (showed 2Mib in a trace), sql::Connection::OpenInternal. These will be the entry points to the main allocation in the code base, and will capture per-component allocations.
2. There seems to be not many trace events in net/ and there seems to be no central location where trace events can capture most of allocations.

~~Problems: Sometimes allocations made inside the trace events are not displayed under the stack trace.~~
<insert prototype, traces and usefulness>

## Category name for allocations

Trace viewer should support 3rd dimension for category-wise view of allocations using heap profiler.

## Location info from PostTask* calls

Whenever tasks are posted in Chrome, a context is added to the task. See base::Location. This gives us the file name and function name of the tasks.

- This is different from the stack trace processing since we are looking at the highest level calls and not the lowest level calls to malloc.
- Tag each allocation using the current task’s posted from filename and function name and add them to the allocation context.

Problems:

- Tasks can be nested. So, need a stack based mechanism for changing the category names.
- The function that posted the current task cannot be added to stack trace directly. This is confusing.
- Most of the IPC message handling will be shown as “ipc” type. But not the actual folder.

### Solutions:

**Option 1**

- Make the StackFrame a struct:

```
struct StackFrame {
  const char* function_name;
  const char* posted_reason;
  const char* category_name;
};
```

- The default category name is set to the category name of the trace event.

- While adding to trace, posted_reason is added by appending a string ("initiated_by %s", posted_reason) before adding the function_name. This will work for nested tasks as well.
- For category name (which is currently the type_name), we take the last category_name and use the folder to categorize.
- For IPC messages I can add posted_reason as the message name (this is available in release builds as well. So for all the memory allocated while handling ipc messages, the category name will be ipc/ and when you open ipc you will see "initiated_by<message name>"

- We cannot add reason on each TRACE_EVENT1 or 2 macros in code. Now this needs a new TRACE_EVENT macro for adding these special stack frames on heap profiler:

  Define a TRACE_TASK_INFO(category_name, function_name, posted_reason). In

  At ipc message handler this can be used as:
  TRACE_TASK_INFO("ipc", "Dispatch", message_name)
  At TRACE_TASK_EXECUTION this can be used as:
  TRACE_TASK_INFO(file_name, "MessageLoop::RunTask", posted_function_name)

**Problems**:
AllocationRegister algorithms need to be changed with the stack frame structs.
Who will own the stack frame objects? This could be a huge change in AllocationRegister.

**Option 2:**

- To keep the change simple we can just have one extra stack of strings for categories and the new trace macro.
- Add the posting function name as a stack frame. This is valid in sense that the task was called because it was posted by the function, so it shows up before the actual stack trace of the task.
  - Can we have a hack to include an extra stack frame "reason" before adding to stack.
  - This can be identified when adding to trace event and merged with the next event in stack trace.
  - Pseudo_stack will look like:
    *ThreadMain -> reason -> posted_function -> RunTask -> functions..*
    Trace will look like:
    *ThreadMain -> initiated_by posted_function -> Runtask - > functions..*
- Push and pop the file name or the ipc message name on the categories stack using the new trace macro.
- The last entry on the category stack is used to categorize allocations.

**Opens:**

1. But, should the ipc message name should be a part of stack frame to reduce noise in category? If so the category would show up as "ipc" which does not mean anything.

2. Should ipc message name be added as type once the support for category is implemented ? What if we have actual object types?
3. Should we change the IPC macro to include the filename along with message name? How much binary size increase could this cost?

### PROs:

- It gives information of which component is to be blamed.
- Covers more than 80% of the malloc usage from experiments.

### CONs:

- It only gives the top level functions. We need to dive deep into these using pseudo stack trace.
- Lot of memory is allocated between and before task start.
- Sometimes threads do not store posted_from, like graph task runner.

### Sample trace:

See full trace file.

## Category from thread names

Yay all thread in chrome has a name. It is much better to see the memory numbers split by thread and get a better idea about it. No traces specifically, but it is quite useful in my experiments, see bugs 571549 and 581368.

This seems really useful since a lot of memory is allocated not in the tasks Eg: BrowserMain startup. Some tasks do not have posted from location information. Eg: worker pool threads. This covers more than 95% of malloc usage accounted in heap profiler.

It is simple to obtain the thread names from tracked_objects. The thread name is added as the first entry in the stack trace.
It is tricky to get the thread names since it will be needed even after the thread dies when the memory allocation lives. ThreadData leaks an instance of thread name string for each thread. But, it does not do so in tests. So, use the ThreadIdNamemanager which also leaks the thread name. This can be added to the top of stack trace.

PROs:

- All the memory in the heap profiler is categorized into a thread. So, we know the origin of the memory. So, easier to track down issues.

CONs:

- The memory doesn't necessarily belong to the thread and ownership could be moved to different threads. But, it still gives some information rather than unknown since trace events are missing.

See traces : trace after closing tabs.

## Adding category name from trace events

This mainly shows very less detail since it only shows the categories enabled by the user. So, he already knows that these are categories. But maybe worth showing a split by category. The trace category of the last event in the pseudo stack is considered as the type.

PROs:

- This is able to categorize 97% of the allocations that were hooked by the shim.
- This can be used along with the task names since all the “toplevel” categories will be classified better with the task locations.

CONs:

- The category sometimes does not make sense, when it shows “toplevel” or “browser”.
- The category of the allocations can just change by adding trace events, so these categories would be vaguely defined.

## Grouping allocations using heap profiler

Looking at the call graph bottom-up. See the major allocators. This will be the major points of convergence.

1. std::allocator
2. skia malloc - sk_malloc, sk_calloc and friends.
3. v8 malloced::new
4. AlignedAlloc (AllocateFFmpegSafeBlock mainly) etc.
5. UncheckedCalloc, UncheckedMalloc
6. IOBuffer::IOBuffer

The problem is we **cannot add trace events** here : will cause a massive number of trace objects and events. Tracing renderer runs out of memory with just a few dumps. But, we can mark these as type-info in the heap profile. At these functions we either know the types or we know some info about the objects.

The notion of type can be modified as an attribute to specify what what the memory is used for, that is it could be the object names or the component name such as “V8_objects”. Such types in the profiler give more information than just the object types.

### How memory allocations are distributed

With different experiments, instrumenting all of the above functions:

| No | Low level function | Allocated Memory | | | Usage % |
|---|---|---|---|---|---|
| | | browser | renderer | browser | renderer |
| 1 | std::allocator::allocate** | 1-20 Mib | 1-6 Mib | 5-25% | 1-10% |
| 2 | sk_malloc and sk_calloc | < 2Mib | 10-20Mib | < 4% | 7-15% |
| 3 | v8::Malloced::New | 1-6Mib | 0-10Mib | 3-9% | 0-15% |
| 4 | base::AlignedAlloc | 0 | 0-30Mib | 0 | 0-25% |
| 5 | UncheckedMalloc | Mostly same as sk_malloc. Only main user is skia. | | | |
| 6 | scoped_ptr, refptr | 1-10Mib | similar | 5-25% | similar |
| 7 | IOBuffer | 2-5Mib | 0 | 3-6% | 0 |
| 8 | UNKNOWN* | 5-40Mib | 10-20Mib | 30-60% | 20-50% |

** The readings for std::allocator is only from the components that have src/ or v8/src/ or third_party/WebKit or third_party/skia in the path when compiling.

Relevant trace files: trace1 , trace2

* What does UNKNOWN consist of:

- This has all other c++ objects allocated in chrome that is not annotated. These are all the small allocations and do not go through the main converging points which allocate memory. They directly call malloc/ new. These can be accounted using the pseudo stack traces (See also Adding more TRACE_EVENT section).
- Third party libraries allocate memory. See 4th point Problems and 1st point in Open questions.

## Skia Object allocations

As we have seen Skia accounts for most of the memory in both browser and renderer. Getting type info of all the skia objects is easy! :)

- Most skia allocations happen via sk_malloc, sk_calloc and sk_realloc.
  All we need to do is have an extra argument in these functions for example:
  `void* sk_malloc_flags(size_t size, unsigned flags, const char* type_info_for_tracing);`

  Number of locations where these functions are used are 75. It is not hard to manually annotate all of these with type info. If not type, a component name that makes sense to skia folks for debugging.

- Other option would be to just label the memory as “skia_objects”, if Skia is not happy with having type names all over their code base.

<What about SkRefCnt? Update more from this internal discussion.>

## Replacing std::allocator

For **gcc** c++ library:

- gcc library defines std::allocator in “bits/allocator.h” file. This file only defines std::allocator class.
- We include a bits/allocator.h in chromium path and redirect all includes to out local file and have custom implementation of std::allocator.
- This new definition of std::allocator will set the type info of each object it allocates.

Opens:

- In stdcxx used in clang llvm, the allocator is defined in “memory” and this file contains various other definitions and it is not worth redefining this whole file.
- No idea about msvc libraries yet.

Note:

1. std::allocator accounts for only 5-15% of the memory usage.
2. The readings are not accurate since there could be other components that do not have src/ in include path which will end up using the other std::allocator.

Top types displayed by std::allocator:

| Object types: | usage in Kib | Description |
|---|---|---|
| ProcessMemoryMaps::VMRegion | ~500 | Used by Tracing. These should be excluded. |
| MemoryAllocatorDump | ~80 | |
| PMD::MemoryAllocatorDumpEdge | ~25 | |
| net::StreamSequencerBuffer::BufferBlock | ~30 | io buffers, useful. |
| unsigned char | 10-5000 | strings?<br>not useful info. |
| int | ~100 | |
| unsigned short | 10-3000 | |
| std::_Rb_tree_node of strings | 50 | No useful info again. |
| scheduler::internal::TaskQueueImpl::Task | ~300 | useful |

| | | |
|---|---|---|
| scoped_refptr<media::StreamParserBuffer> | 10-6000 | On youtube pages. Useful. |
| cc::TransferableResource | 10-20 | Useful |

Examples can be seen in this trace1 and trace2

Usually these nodes are in range 20-100Kib, which accounts for 0.3% each.

The actual object types. But this seems to have mainly int, short, long and char and tracing overheads.
We have 2 options to display types from std::allocator:

1. All objects allocated by std::allocator can be shown as “std_allocated_objects”.
2. Each object type info is added to trace. The type is extracted from the function name as done in PartitionAlloc profiler. This can only be done for non-official builds due to chrome binary size increase.
3. We can group all the basic types into single entity as “basic_types” and have other types. Even if we did that these objects show up as “other”.

#### PROs:

- This could be useful when cases like one object type is created too many times.
- It also tells us what amount of memory is allocated using std::allocator. This is useful in cases where the memory is dark and we don’t know who is allocating the memory. If it is marked as std::allocator, then we know that these are all C++ objects used in chrome rather than memory malloced directly.

#### CONs:

- It is really hard work to replace std::allocator across all platforms/ compiler uniformly, since there is no standard construct to do it.
- Current hacks are difficult to upstream.

## Types from scoped_ptr and scoped_refptr

It is straightforward to obtain object types from scoped_ptr just like PartitionAlloc type info. This one covered most of the C++ objects in the process. Surprisingly it comes out to be a very low percent of the total.

#### PROs:

- Getting the type info is very easy.
- Useful when spotting issues like some object type is leaked.

#### CONs:

- The major portion doesn’t have type info.
- More often memory allocator allocates large spaces for containers a member of a class and the memory marked by scoped_ptr is in middle of large malloc segment.

### V8 and other malloc-like functions in Chrome

We have other malloc like functions that are used in chrome:

a. **v8 New()** can be marked as “v8_objects” in general. It might not be worth adding extra argument here. We could just group them as V8 objects and let the V8 folks do hard work if there’s regression.

b. **UncheckedMalloc, Calloc**. These are mainly used by skia. But I have noticed some cases that are not covered in skia. It is always better to have it complete, so add type argument here as well.

c. Set types at **IOBuffer constructor** and allocation sites. One might argue that why not have a dump provider for this. But there are too many IOBuffers allocated in the process and we do not want all these dump providers hanging around for accounting 3-5% of the memory. There are also many paths that can store io buffers depending on the socket connection. So, it is not easy to get a dump provider for it.

d. Adding extra argument to **base::AlignedAlloc** as:
void* AlignedAlloc(size_t size, size_t alignment, const char* type_name)

There are only 30 places that use AlignedAlloc. This would give us more details about media, audio, video, image buffers that use AlignedAlloc, especially mpeg decoder.

#### PROs:

- It gives little more information of which component is to be blamed

#### CONs:

- It is not easy to maintain. There can be new functions of interest and that would not be instrumented. But this is same problem as the dump providers, there can always be new allocators in chrome. At the same time, the refactoring of code happens more frequently, so it is more likely that these instrumentations can be obsolete.

### Current minor issues with experimentation:

- The malloc total given by mallinfo doesn’t match with the value from the heap profiler. <primiano’s upstream version will probably have a fix>
- I am using a old heap profiler from Tcmalloc to find the allocating functions.
- I am experimenting only on Linux platform.

## Clang hack for stack trace

Dmitri made a profiler that collects native stack traces without changing clang, but overall idea is to add instrumentation automatically, which will involve clang plugins or something

similar. ~~hack to clang~~ Process the stack trace to get the lowest level component that allocated memory.

This gives the component that allocated the memory and the data seems very useful.
Results in this doc.
But, this cannot be upstreamed on normal builds since it's a clang change and also this hack works only on clang, not gcc and msvc.
This can act as the ground truth and we will try to get as close as possible to this data.
Can it be done using the stack traces from Primiano's doc.

## Open questions

1. Major usages not covered by the above approach are from:
    a. sqlite(3%) - accounted in trace as dump provider.
    b. leveldb(3%) - accounted in trace as dump provider.
    c. libglib(6%) - starts before chrome starts. Cannot even have trace events for these allocations. It can be a significant number and it varies with similar libraries on other platforms.
    d. pl_arena(9%) - does anyone know what this is? It is used by net/. But I do not get stack traces. It sometimes accounts for a lot of memory.
    e. libnss (2-3%) - ideas welcome.
    f. and more libraries.
    *Note: The % mentioned were maximum numbers seen.*

The only information we can show about these are the entry point function in the pseudo stack trace. But there are too many functions that allocate memory in leveldb and sqlite, and the libglib starts before chrome, so it is hard to add trace macros to entry point of the libraries. These account for 20 - 30% of the memory usage in browser.
We capture some of these in dump providers, like leveldb and sqlite. But there is still major portion of it left unknown in tracing. How do we capture this part?

2. How do we correlate the stats from heap profiler and dump providers. We saw that few libraries cannot be accounted in heap profiler and only shown in dump provider list. Should this be left unknown or should we try to correlate and make it smart?

3. How do we choose which way to categorize memory or are we going to have different views for each type of view. We can categorize using task locations, trace event categories or thread names.

4. Redefining std::allocator is possible in gnu_cxx by replacing the allocator.h file in chromium include path. But in stdcxx for llvm, the std::allocator is defined in memory file. This also defines various other constructs. Replacing this file would lead to redefining of all these in memory.
How can we hook std::allocator otherwise?

5. Similarly Dmitri's profiler can work in clang only. Should we upstream it for developers only? behind build flag? Can it be done using the stack traces from Primiano's doc.

# Pending investigations / discussions

- ~~Solve the mystery above: "Sometimes allocations made inside the trace events are not displayed under the stack trace"~~
- Is it worth instrumenting scoped_ptr / scoped_refpr to get typeinfos? Which % of types (w.r.t memory seen by the malloc heap profiler) would we get?
- Is it worth instrumenting std::allocator? Which % of memory would it cover? Which types would it give?
- How do we instrument net/ ? This includes IOBuffer, PLArena and libnss mentioned above. - Discussed with few networking folks and seems very difficult to find a common point where allocations diverge or converge. So can't add dump providers or trace events or type_inof context hacks.
- See how much coverage we get using FROM_HERE Location info from PostTask calls and thread names.
- Category names from trace events.

# Appendix

## Sample traces with object types

1. Detailed types from std::allocator
2. News pages scrolled (verge, latimes, nytimes)
3. Videos and News (youtube, verge)
4. Trace with types as component names from task location.
5. Traces with components from clang stack trace: trace1 and trace2.
6. Trace with post task location and thread names.

## Sample call graphs which were useful to write this doc

These are just one sample of each type and I used many of these to find out all the allocation points mentioned in this doc.

- malloc graph
- mmaps graph