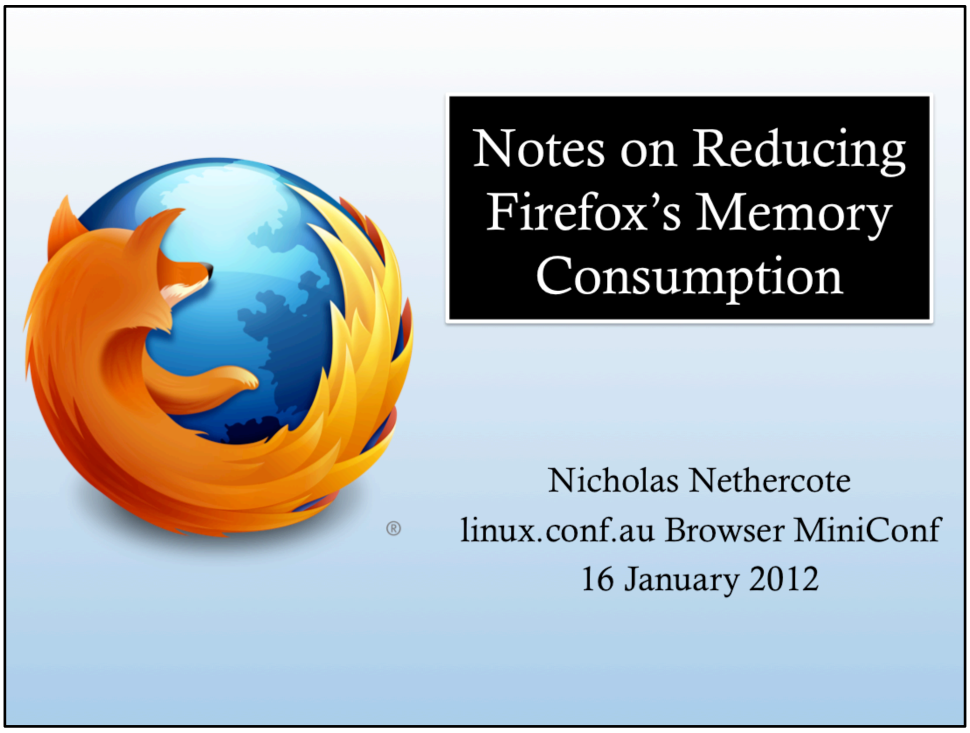


Notes on Reducing Firefox's Memory Consumption

Nicholas Nethercote
linux.conf.au Browser MiniConf
16 January 2012

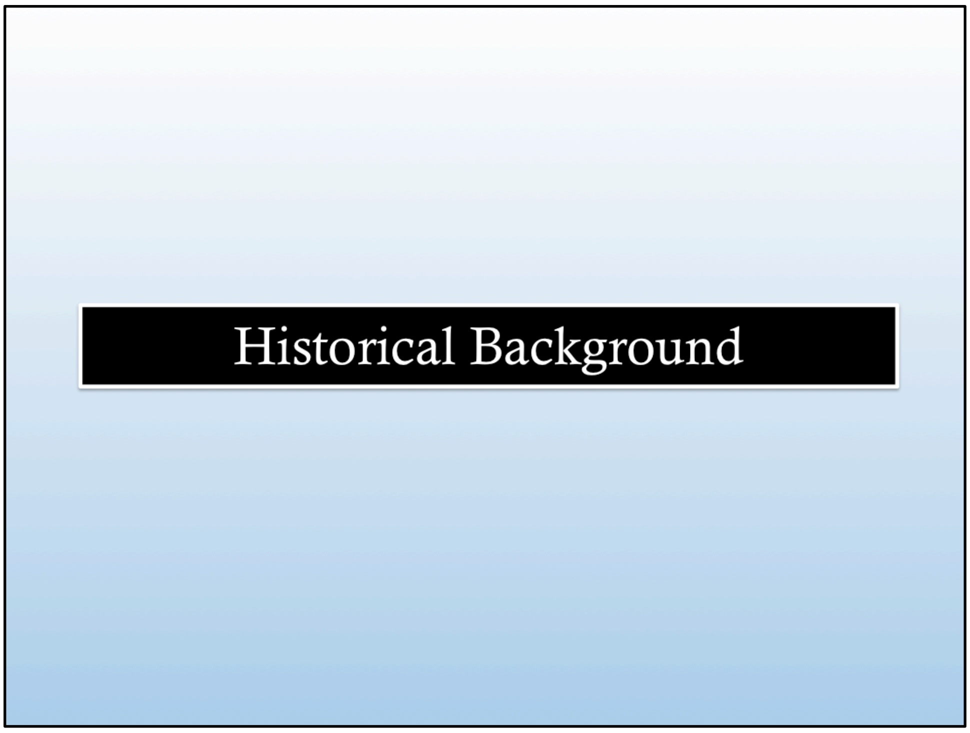
Notes to readers:

- This PDF contains both slides and notes. The notes contain most of the content. This talk was created with PowerPoint, which has a shortcoming: if the notes for a single slide do not fit on a single page when printed, the excess notes are simply cut off. For this reason, in this PDF I have duplicated several slides in order to split their notes across two pages.
- Some slides feature red boxes. In the live version of the talk I used PowerPoint animations to show these red boxes one at a time, in order to point out particular details on each slide. But in this PDF all the red boxes are visible at once. You'll just have to imagine the animation when reading.

I'm going to start by telling you a secret.

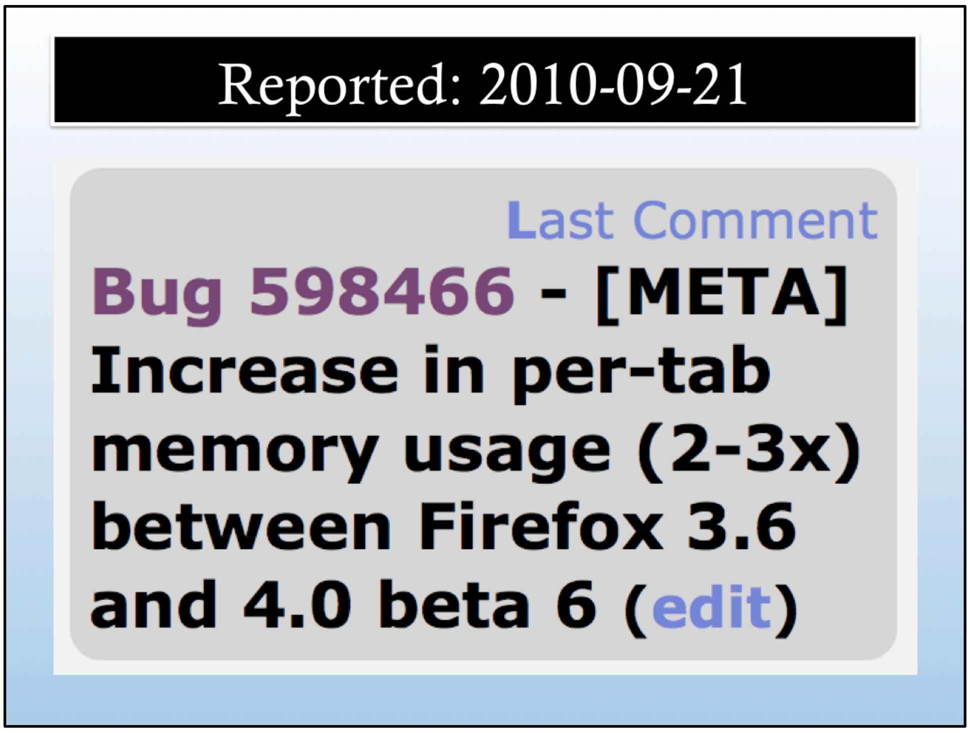
Firefox has a reputation

- That secret is that Firefox has a reputation… for using a lot of memory.
- In fact, I have two rules of thumb.
- The first is that any online discussion of web browsers will degenerate into an argument about which browser is best.
- The second is that any argument about which browser is best will feature at least one complaint about Firefox's memory consumption.
- "Have they fixed the memory leak yet?" is a common question.
- In fact it's so common that I decided to do something about it. And that's what this talk is about.



Historical Background

- So this reputation is partly deserved, but partly undeserved, and its accuracy has varied over the years.
- Firefox 2 and 3 predate my time at Mozilla, but I've heard that Firefox 2 was full of leaks, and as a result, a lot of effort went into Firefox 3 and it was much better. And versions 3.5 and 3.6 were also fairly good. The story is complicated greatly by add-ons, and that's something I'll return to later, but basically the whole 3 series was pretty good.
- And then came Firefox 4. Firefox 4 had a long, difficult gestation. It had a lot of new features. There were 13 beta releases. It took almost 6 months longer than planned.



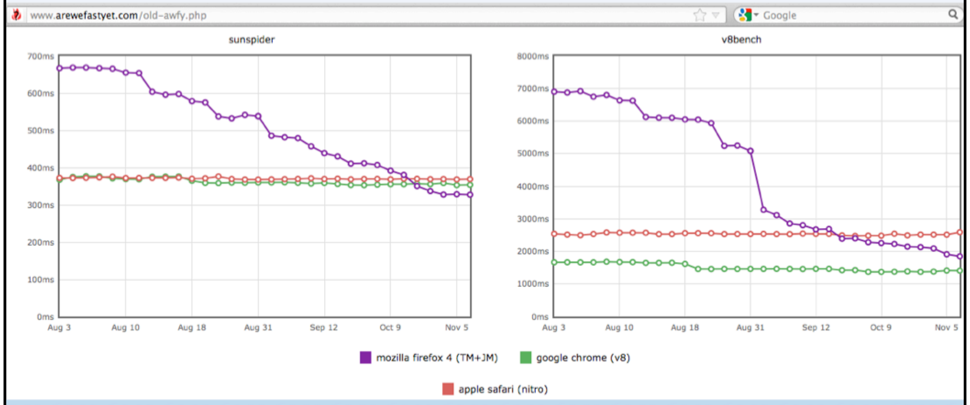
Reported: 2010-09-21

Last Comment

Bug 598466 - [META] Increase in per-tab memory usage (2-3x) between Firefox 3.6 and 4.0 beta 6 (edit)

- Partway through its development, it became clear that Firefox 4's memory usage was appalling, due to a combination of leaks and inefficiencies in new code.
- Bug 598466 tracked these problems, and it ended depending on over 50 different bugs. A lot of them were fixed by the time Firefox 4 was released in March last year – I fixed a few of them myself -- and yet Firefox 4 still used significantly more memory than Firefox 3.6.

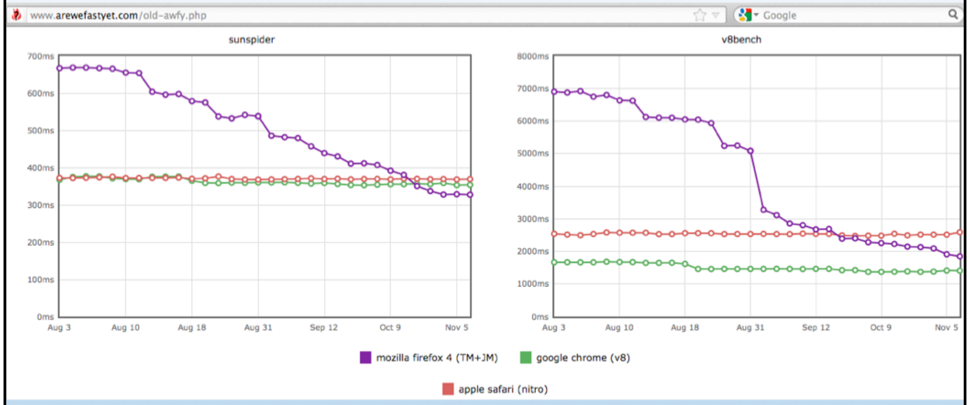
Why was FF4 bad?



- JS engine: JaegerMonkey, fatvals, GC
- `image.mem.min_discard_timeout_ms`
- `innerHTML` setting leak

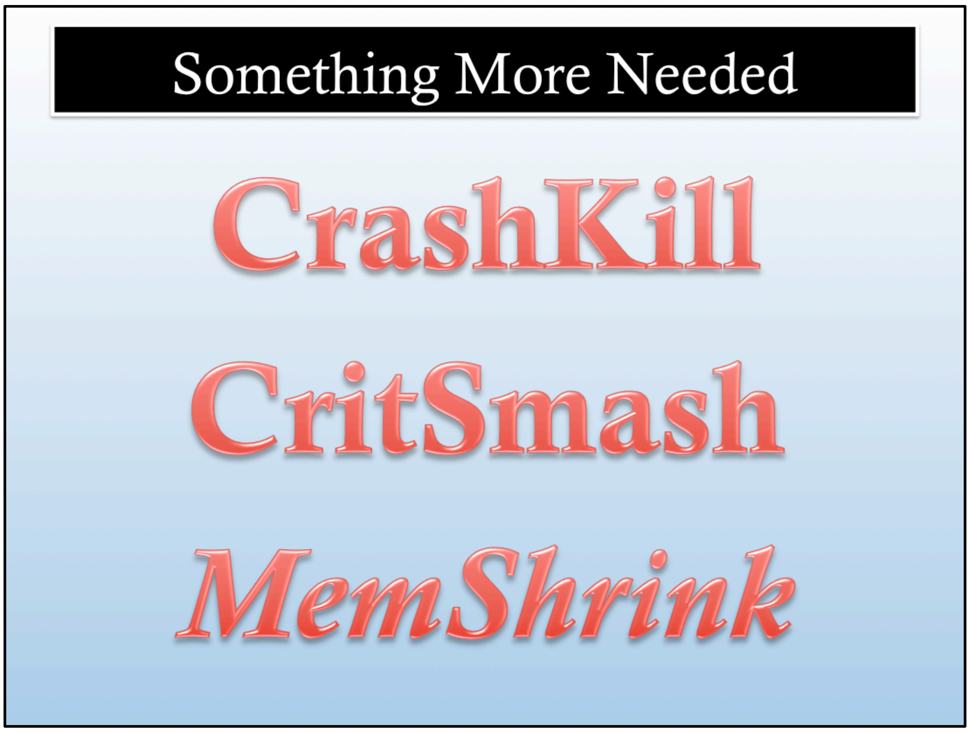
- There are several reasons why Firefox 4 was so memory hungry.
- The first one had to do with the JavaScript engine. Firefox 4 featured a new JavaScript JIT called JaegerMonkey, which did a fantastic job of catching up with Chrome and Safari on the two most widely-used benchmarks, SpiderMonkey and V8. The graphs on this slide show the progress made from the second half of 2010.
- However, memory consumption was not given much attention during JaegerMonkey's development. As a result JaegerMonkey generates quite fast code, but also a lot of code – multiple MBs of it for many websites.
- Another big change in the JavaScript engine was that the fundamental representation of a JavaScript value changed from 32-bits to 64-bits. This new representation, called fatvals, is faster to work with, but obviously takes up more memory.
- Finally, the garbage collection heuristics in Firefox 4 were terrible. Well, they worked beautifully for Sunspider and V8, but not so well with normal websites. In particular, the garbage collector just didn't run often enough, and people who left Firefox open overnight often came back to an unresponsive machine in the morning. This problem was finally fixed in Firefox 7.

Why was FF4 bad?



- JS engine: JaegerMonkey, fatvals, GC
- `image.mem.min_discard_timeout_ms`
- `innerHTML` setting leak

- Moving beyond the JavaScript engine, there were two other big problems.
- First, there were some changes to the handling of images. In order to view an image, you must decompress it into a raw pixel format, which is much larger than the original compressed image. Browsers have heuristics for deciding when to decompress images and when to discard the decompressed data. If you opened a webpage in a background tab, Firefox 4 would decompress all the images within it, and if you didn't switch to that tab, it would discard the data only after 2 or 3 minutes. If you opened lots of pages in background tabs it could send memory usage through the roof. That heuristic has since been improved so that images in background tabs are not decompressed until that tab gains focus, and if you switch away, the decompressed data is discarded after only 10 or 20 seconds.
- Finally, Firefox 4 had a new HTML5 parser. It had a bug which meant that every time you set an innerHTML property on an element, some memory would leak. And that's a pretty common operation on many websites. This was fixed in Firefox 5.



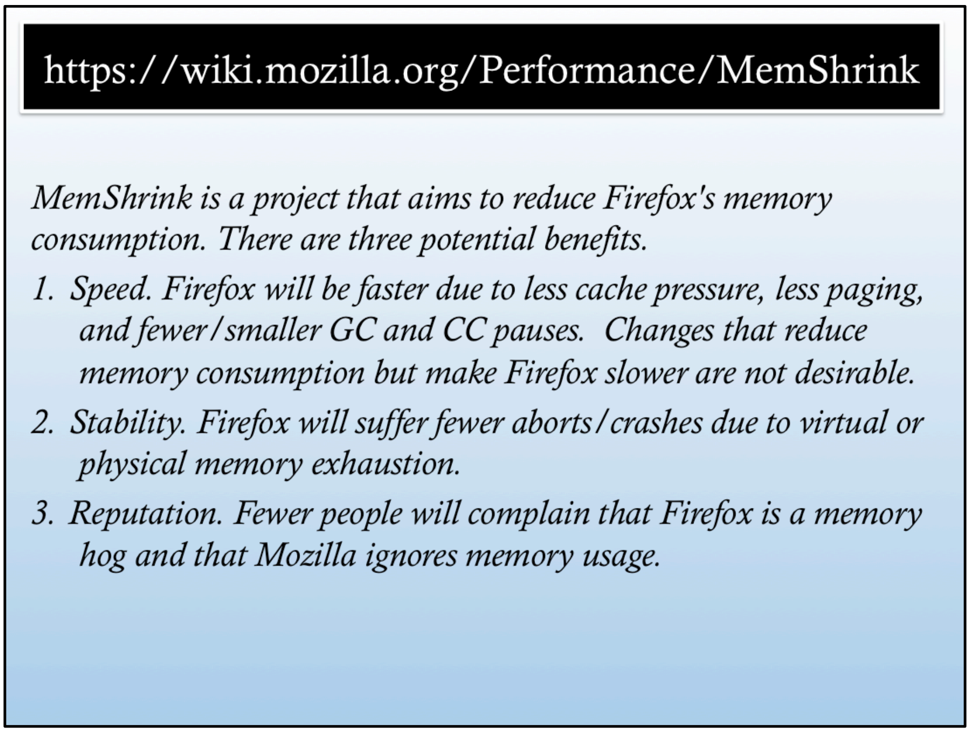
Something More Needed

CrashKill

CritSmash

MemShrink

- Mozilla had previously had some internal projects to improve specific aspects of Firefox.
- One project called CrashKill focused solely on reducing how often Firefox crashes.
- Another project called CritSmash focused solely on reducing the number of critical security vulnerabilities.
- Both of those projects had people dedicated to them, with weekly meetings that tracked bugs and general progress, and they'd both worked well.
- Even before Firefox 4 was released, it was clear a similar project focused on memory consumption was needed, and "MemShrink" was an obvious name for it. I started this project in March last year, but didn't really get off the ground until June, when we began having weekly meetings.

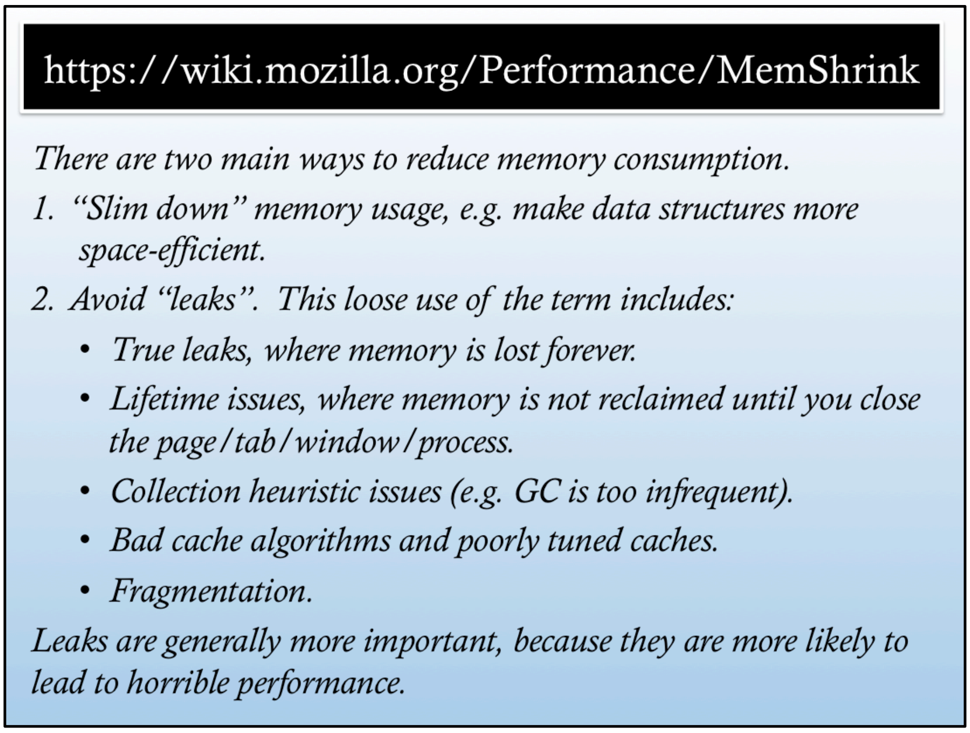


https://wiki.mozilla.org/Performance/MemShrink

MemShrink is a project that aims to reduce Firefox's memory consumption. There are three potential benefits.

1. *Speed. Firefox will be faster due to less cache pressure, less paging, and fewer/smaller GC and CC pauses. Changes that reduce memory consumption but make Firefox slower are not desirable.*
2. *Stability. Firefox will suffer fewer aborts/crashes due to virtual or physical memory exhaustion.*
3. *Reputation. Fewer people will complain that Firefox is a memory hog and that Mozilla ignores memory usage.*

- I'm now going to read the introduction from the wiki page for the MemShrink project, because it explains nicely what the project is all about.



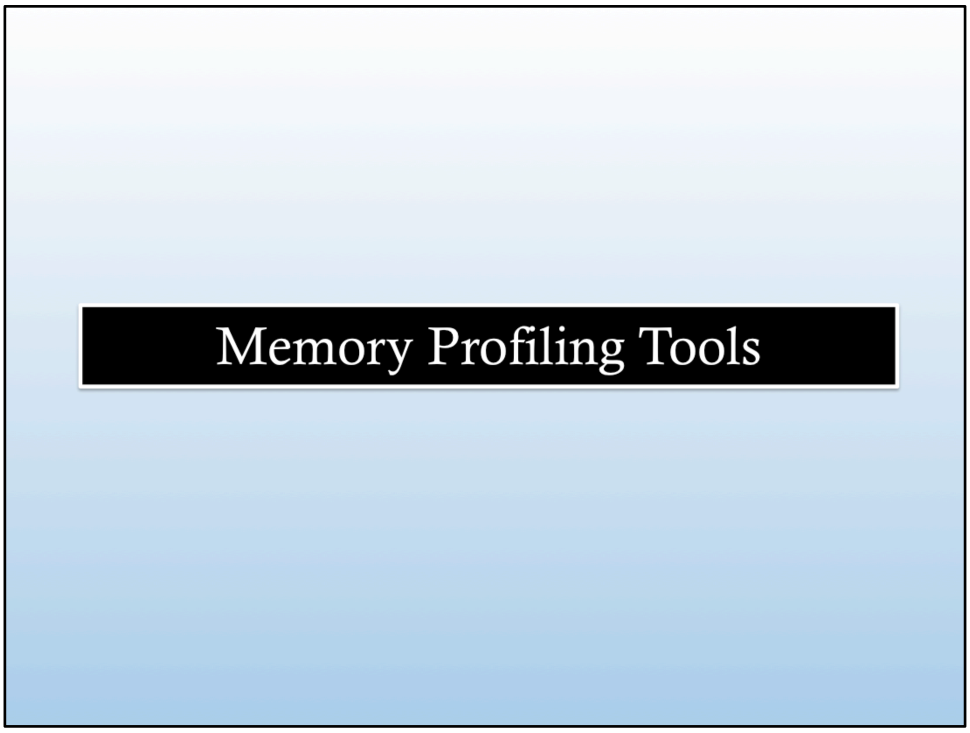
https://wiki.mozilla.org/Performance/MemShrink

There are two main ways to reduce memory consumption.

1. *"Slim down" memory usage, e.g. make data structures more space-efficient.*
2. *Avoid "leaks". This loose use of the term includes:*
   - *True leaks, where memory is lost forever.*
   - *Lifetime issues, where memory is not reclaimed until you close the page/tab/window/process.*
   - *Collection heuristic issues (e.g. GC is too infrequent).*
   - *Bad cache algorithms and poorly tuned caches.*
   - *Fragmentation.*

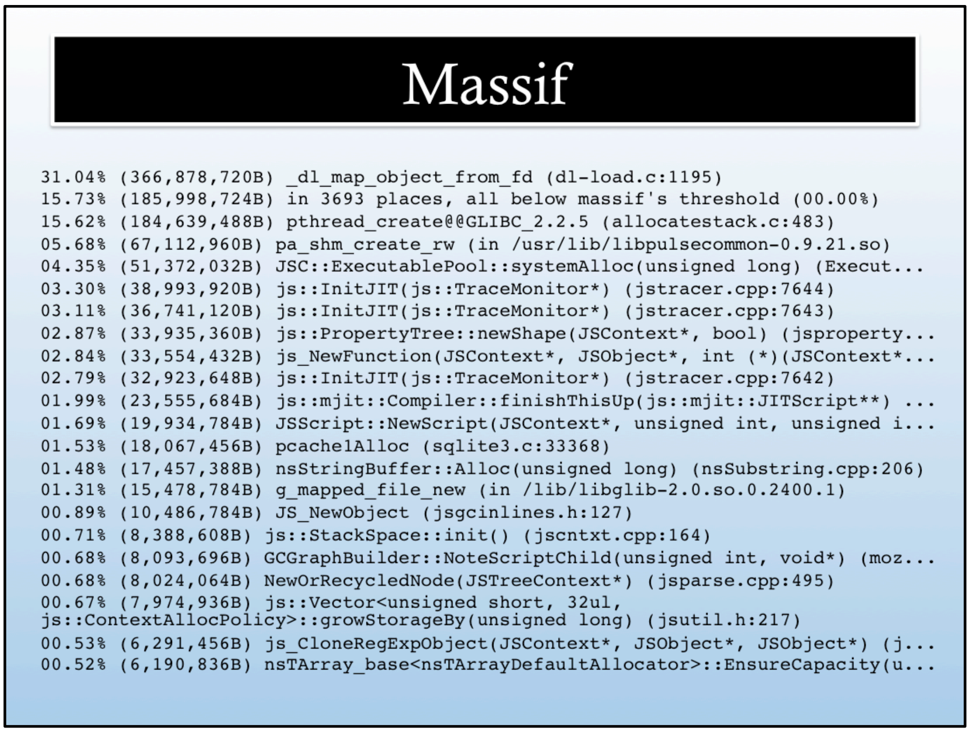
Leaks are generally more important, because they are more likely to lead to horrible performance.

- So that's the aims and goals of MemShrink.



Memory Profiling Tools

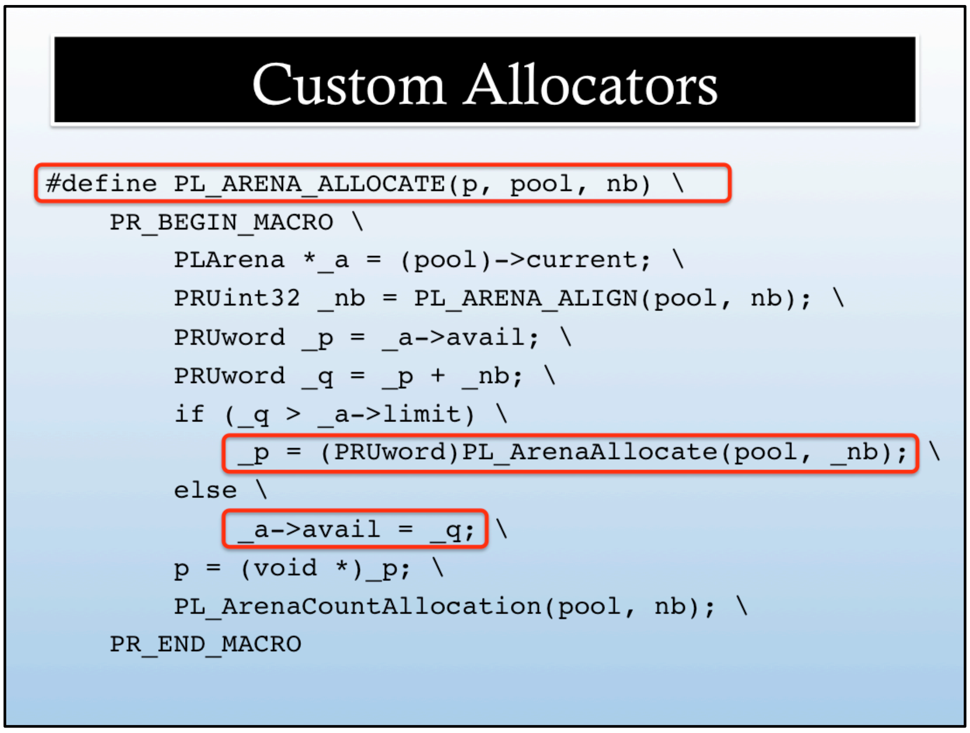
- But before you can reduce memory consumption you need to understand it.
- And that requires tools.
- Unfortunately, good memory profiling tools are hard to find.



Massif

```
31.04% (366,878,720B) _dl_map_object_from_fd (dl-load.c:1195)
15.73% (185,998,724B) in 3693 places, all below massif's threshold (00.00%)
15.62% (184,639,488B) pthread_create@@GLIBC_2.2.5 (allocatestack.c:483)
05.68% (67,112,960B) pa_shm_create_rw (in /usr/lib/libpulsecommon-0.9.21.so)
04.35% (51,372,032B) JSC::ExecutablePool::systemAlloc(unsigned long) (Execut...
03.30% (38,993,920B) js::InitJIT(js::TraceMonitor*) (jstracer.cpp:7644)
03.11% (36,741,120B) js::InitJIT(js::TraceMonitor*) (jstracer.cpp:7643)
02.87% (33,935,360B) js::PropertyTree::newShape(JSContext*, bool) (jsproperty...
02.84% (33,554,432B) js_NewFunction(JSContext*, JSObject*, int (*)(JSContext*...
02.79% (32,923,648B) js::InitJIT(js::TraceMonitor*) (jstracer.cpp:7642)
01.99% (23,555,684B) js::mjit::Compiler::finishThisUp(js::mjit::JITScript**) ...
01.69% (19,934,784B) JSScript::NewScript(JSContext*, unsigned int, unsigned i...
01.53% (18,067,456B) pcache1Alloc (sqlite3.c:33368)
01.48% (17,457,388B) nsStringBuffer::Alloc(unsigned long) (nsSubstring.cpp:206)
01.31% (15,478,784B) g_mapped_file_new (in /lib/libglib-2.0.so.0.2400.1)
00.89% (10,486,784B) JS_NewObject (jsgcinlines.h:127)
00.71% (8,388,608B) js::StackSpace::init() (jscntxt.cpp:164)
00.68% (8,093,696B) GCGraphBuilder::NoteScriptChild(unsigned int, void*) (moz...
00.68% (8,024,064B) NewOrRecycledNode(JSTreeContext*) (jsparse.cpp:495)
00.67% (7,974,936B) js::Vector<unsigned short, 32ul,
js::ContextAllocPolicy>::growStorageBy(unsigned long) (jsutil.h:217)
00.53% (6,291,456B) js_CloneRegExpObject(JSContext*, JSObject*, JSObject*) (j...
00.52% (6,190,836B) nsTArray_base<nsTArrayDefaultAllocator>::EnsureCapacity(u...
```

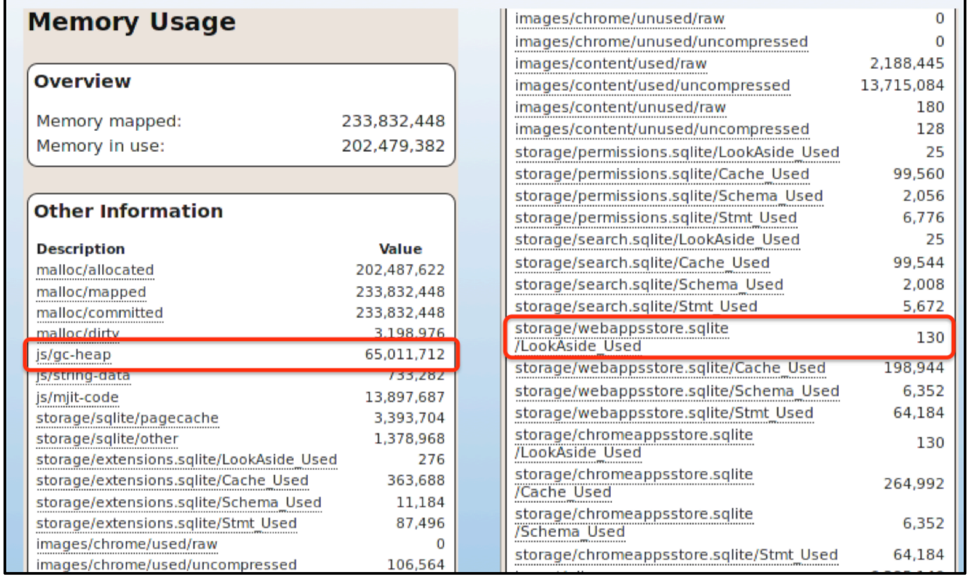
- One tool I used with some success for Firefox 4 is a Valgrind tool I wrote a few years ago called Massif.
- Massif's main strength is that it gives very fine-grained data about memory allocations. The output shown on this slide is only a tiny fraction of what you get for a single run.
- However, Massif has several weaknesses.
- First, it's extremely slow.
- Second, because it's a Valgrind tool it doesn't work on Windows.
- Third, its output is entirely text, which I personally like but plenty of people don't. Also, there's so much output that it's easy to get lost if you're not used to it.
- But most importantly, it suffers from a problem that all general-purpose memory profilers have with Firefox.


Custom Allocators

```
#define PL_ARENA_ALLOCATE(p, pool, nb) \
    PR_BEGIN_MACRO \
        PLArena *_a = (pool)->current; \
        PRUint32 _nb = PL_ARENA_ALIGN(pool, nb); \
        PRUword _p = _a->avail; \
        PRUword _q = _p + _nb; \
        if (_q > _a->limit) \
            _p = (PRUword)PL_ArenaAllocate(pool, _nb); \
        else \
            _a->avail = _q; \
        p = (void *)_p; \
        PL_ArenaCountAllocation(pool, nb); \
    PR_END_MACRO
```

- That problem is that Firefox has numerous custom allocators that are layered on top of malloc.
- One good example is what's called a PLArenaPool, which is an allocator that lets you allocate lots of objects individually but then free them all in a single operation.
- Imagine that Firefox makes 10 calls to the PL_ARENA_ALLOCATE macro.
- The first time it will call the PL_ArenaAllocate function, which allocates a large chunk with malloc
- The next 9 times might be satisfied from that same chunk, and not require any additional calls to malloc.
- Any general-purpose memory profiler will blame the first call for all the allocations, and the other nine won't be blamed at all.
- Firefox does enough custom allocations like this that Massif's output gets confusing and misleading.

about:memory (Firefox 4)



Memory Usage

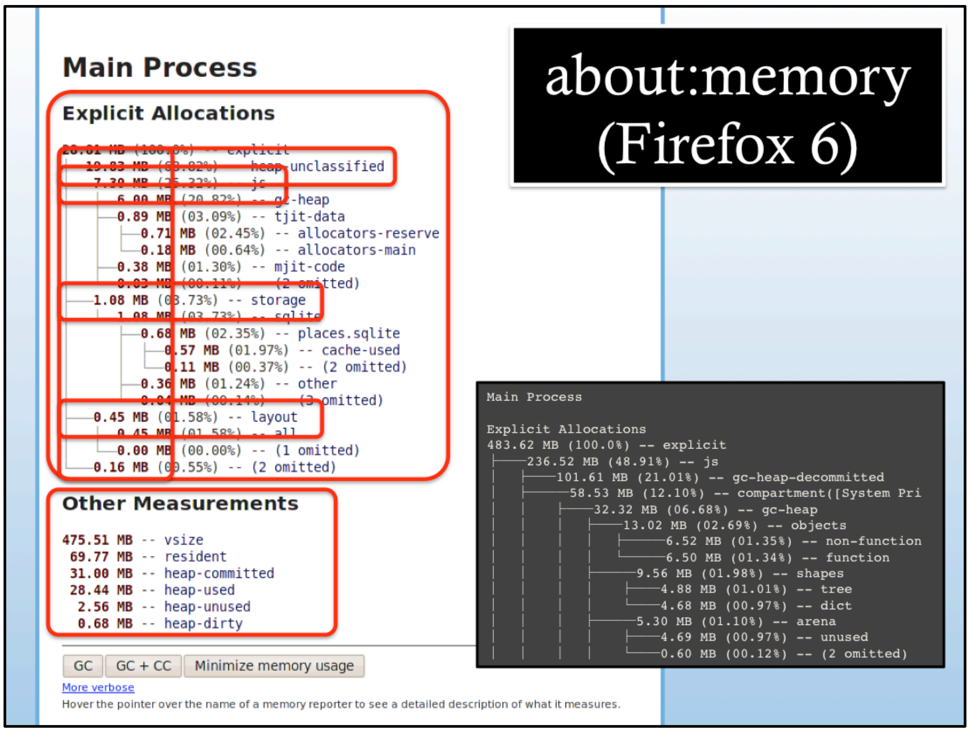
Overview

Memory mapped:	233,832,448
Memory in use:	202,479,382

Other Information

Description	Value
malloc/allocated	202,487,622
malloc/mapped	233,832,448
malloc/committed	233,832,448
malloc/dirty	3,198,976
js/gc-heap	65,011,712
js/string-data	733,282
js/mjit-code	13,897,687
storage/sqlite/pagecache	3,393,704
storage/sqlite/other	1,378,968
storage/extensions.sqlite/LookAside_Used	276
storage/extensions.sqlite/Cache_Used	363,688
storage/extensions.sqlite/Schema_Used	11,184
storage/extensions.sqlite/Stmt_Used	87,496
images/chrome/used/raw	0
images/chrome/used/uncompressed	106,564
images/chrome/unused/raw	0
images/chrome/unused/uncompressed	0
images/content/used/raw	2,188,445
images/content/used/uncompressed	13,715,084
images/content/unused/raw	180
images/content/unused/uncompressed	128
storage/permissions.sqlite/LookAside_Used	25
storage/permissions.sqlite/Cache_Used	99,560
storage/permissions.sqlite/Schema_Used	2,056
storage/permissions.sqlite/Stmt_Used	6,776
storage/search.sqlite/LookAside_Used	25
storage/search.sqlite/Cache_Used	99,544
storage/search.sqlite/Schema_Used	2,008
storage/search.sqlite/Stmt_Used	5,672
storage/webappsstore.sqlite /LookAside_Used	130
storage/webappsstore.sqlite/Cache_Used	198,944
storage/webappsstore.sqlite/Schema_Used	6,352
storage/webappsstore.sqlite/Stmt_Used	64,184
storage/chromeappsstore.sqlite /LookAside_Used	130
storage/chromeappsstore.sqlite /Cache_Used	264,992
storage/chromeappsstore.sqlite /Schema_Used	6,352
storage/chromeappsstore.sqlite/Stmt_Used	64,184

- To avoid this problem, we really need the profiling to happen within Firefox. As it happens, exactly such a feature was added in Firefox 3.5. If you visited the page called "about:memory", Firefox would take some internal measurements and display them.
- The really nice thing about this is that it's incredibly easy to use and it's available all the time, even in normal releases.
- Unfortunately, as of Firefox 4, the measurements made in about:memory were pretty much rubbish. It just showed a list of reports; they were mostly very coarse-grained, except for a few that were ridiculously fine-grained; it only reported on some parts of the browser; it was hard to know how much memory wasn't being measured; several of the reported measurements overlapped in non-obvious ways.
- There just wasn't much in the way of useful actions you could take as a result of looking at it.


about:memory (Firefox 6)

Main Process

Explicit Allocations

```
28.81 MB (100.0%) -- explicit
├──19.83 MB (68.82%) -- heap-unclassified
├───7.30 MB (25.32%) -- js
│   ├──6.00 MB (20.82%) -- gc-heap
│   ├──0.89 MB (03.09%) -- tjit-data
│   │  ├──0.71 MB (02.45%) -- allocators-reserve
│   │  └──0.18 MB (00.64%) -- allocators-main
│   ├──0.38 MB (01.30%) -- mjit-code
│   └──0.03 MB (00.11%) -- (2 omitted)
├───1.08 MB (03.73%) -- storage
│   └──1.08 MB (03.73%) -- sqlite
│      ├──0.68 MB (02.35%) -- places.sqlite
│      │  ├──0.57 MB (01.97%) -- cache-used
│      │  └──0.11 MB (00.37%) -- (2 omitted)
│      ├──0.36 MB (01.24%) -- other
│      └──0.04 MB (00.14%) -- (3 omitted)
├───0.45 MB (01.58%) -- layout
│   ├──0.45 MB (01.58%) -- all
│   └──0.00 MB (00.00%) -- (1 omitted)
└───0.16 MB (00.55%) -- (2 omitted)
```

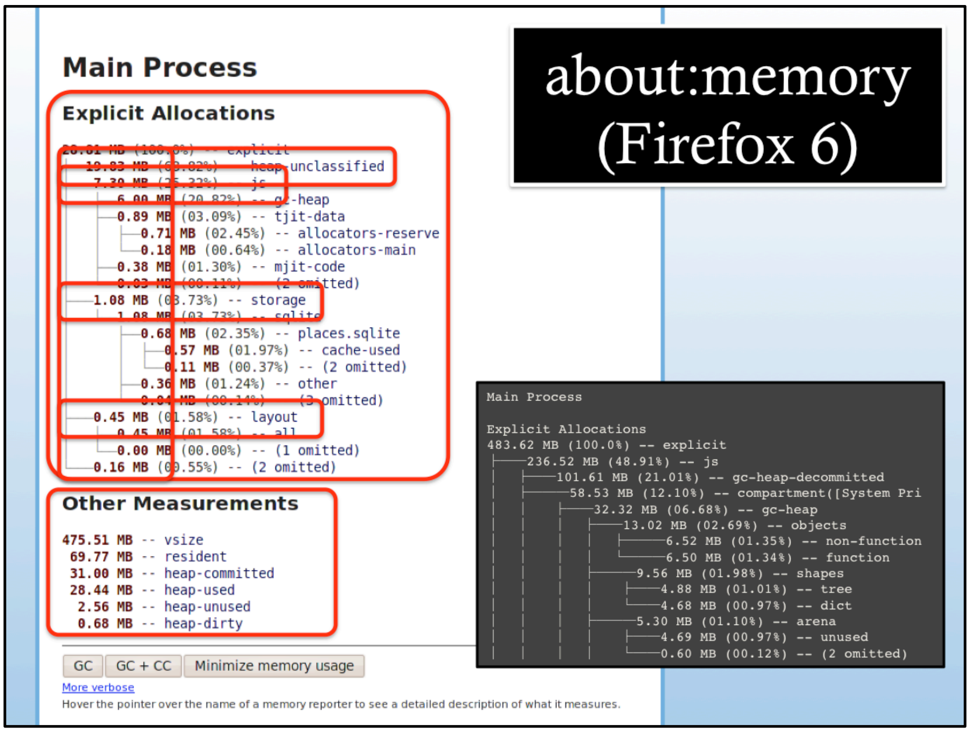
Other Measurements

```
475.51 MB -- vsize
 69.77 MB -- resident
 31.00 MB -- heap-committed
 28.44 MB -- heap-used
  2.56 MB -- heap-unused
  0.68 MB -- heap-dirty
```

Hover the pointer over the name of a memory reporter to see a detailed description of what it measures.

```
Main Process

Explicit Allocations
483.62 MB (100.0%) -- explicit
├───236.52 MB (48.91%) -- js
│    ├───101.61 MB (21.01%) -- gc-heap-decommitted
│    ├─────58.53 MB (12.10%) -- compartment([System Pri
│    │     ├────32.32 MB (06.68%) -- gc-heap
│    │     │    ├────13.02 MB (02.69%) -- objects
│    │     │    │    ├──────6.52 MB (01.35%) -- non-function
│    │     │    │    └──────6.50 MB (01.34%) -- function
│    │     │    ├──────9.56 MB (01.98%) -- shapes
│    │     │    │      ├────4.88 MB (01.01%) -- tree
│    │     │    │      └────4.68 MB (00.97%) -- dict
│    │     │    ├──────5.30 MB (01.10%) -- arena
│    │     │    │      ├────4.69 MB (00.97%) -- unused
│    │     │    │      └────0.60 MB (00.12%) -- (2 omitted)
```

- So I rewrote it and the new version first appeared in Firefox 6.
- The main feature is a tree called "explicit allocations" which covers all the memory that Firefox allocates on the heap and some of the allocations that Firefox does directly with mmap. It doesn't include things like code and data sections, because they're usually of less interest.
- Because it's a tree, you can easily see both coarse-grained and fine-grained views of the data, and its obvious how much memory each browser component is using. In this example the JavaScript engine is using 25%, storage is using 3.7%, and layout is using 1.5%.
- All the unattributed heap allocations fall into a single bucket called "heap-unclassified". We call this unknown stuff "dark matter". The coverage at that stage was terrible, 69% in this example, and I'll talk more about dark matter more shortly.
- Below the tree is a section that contains other measurements that cross-cut the explicit allocations, such as the amount of virtual and physical memory used.



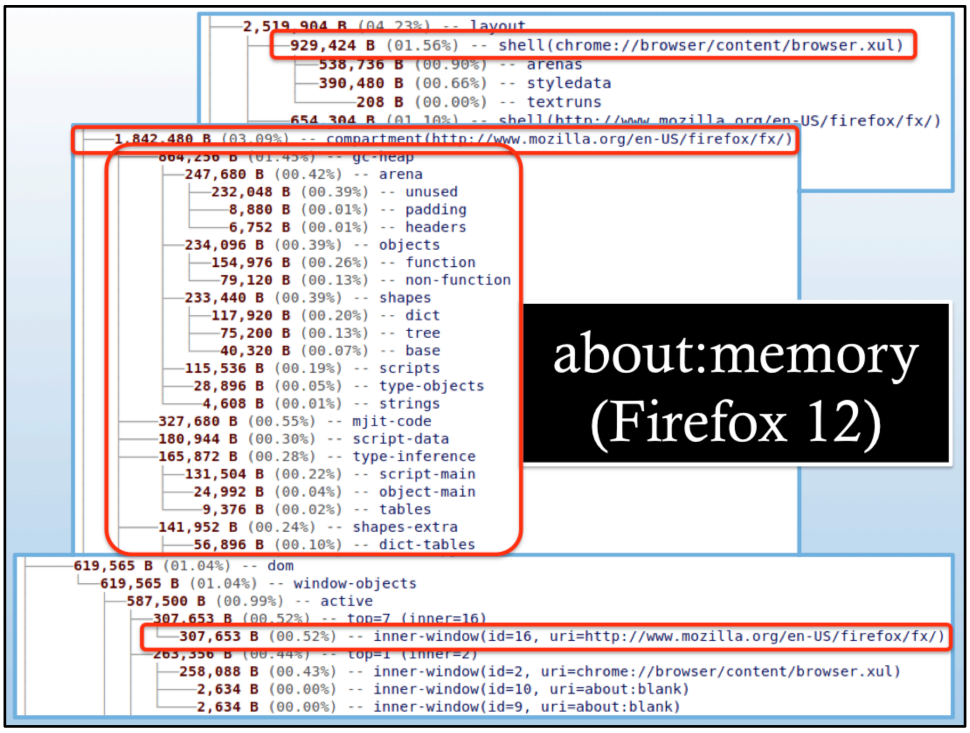
about:memory (Firefox 6)

```
Main Process

Explicit Allocations

28.81 MB (100.0%) -- explicit
├──19.83 MB (68.82%) -- heap-unclassified
├───7.30 MB (25.32%) -- js
│   ├──6.00 MB (20.82%) -- gc-heap
│   ├──0.89 MB (03.09%) -- tjit-data
│   │  ├──0.71 MB (02.45%) -- allocators-reserve
│   │  └──0.18 MB (00.64%) -- allocators-main
│   ├──0.38 MB (01.30%) -- mjit-code
│   └──0.03 MB (00.11%) -- (2 omitted)
├───1.08 MB (03.73%) -- storage
│   └──1.08 MB (03.73%) -- sqlite
│      ├──0.68 MB (02.35%) -- places.sqlite
│      │  ├──0.57 MB (01.97%) -- cache-used
│      │  └──0.11 MB (00.37%) -- (2 omitted)
│      ├──0.36 MB (01.24%) -- other
│      └──0.04 MB (00.14%) -- (3 omitted)
├───0.45 MB (01.58%) -- layout
│   ├──0.45 MB (01.58%) -- all
│   └──0.00 MB (00.00%) -- (1 omitted)
└───0.16 MB (00.55%) -- (2 omitted)

Other Measurements

475.51 MB -- vsize
 69.77 MB -- resident
 31.00 MB -- heap-committed
 28.44 MB -- heap-used
  2.56 MB -- heap-unused
  0.68 MB -- heap-dirty
```

Hover the pointer over the name of a memory reporter to see a detailed description of what it measures.

```
Main Process

Explicit Allocations
483.62 MB (100.0%) -- explicit
├───236.52 MB (48.91%) -- js
│    ├───101.61 MB (21.01%) -- gc-heap-decommitted
│    ├─────58.53 MB (12.10%) -- compartment([System Pri
│    │     ├───32.32 MB (06.68%) -- gc-heap
│    │     │    ├───13.02 MB (02.69%) -- objects
│    │     │    │    ├──────6.52 MB (01.35%) -- non-function
│    │     │    │    └──────6.50 MB (01.34%) -- function
│    │     │    ├──────9.56 MB (01.98%) -- shapes
│    │     │    │    ├────4.88 MB (01.01%) -- tree
│    │     │    │    └────4.68 MB (00.97%) -- dict
│    │     │    ├──────5.30 MB (01.10%) -- arena
│    │     │    │    ├────4.69 MB (00.97%) -- unused
│    │     │    │    └────0.60 MB (00.12%) -- (2 omitted)
```

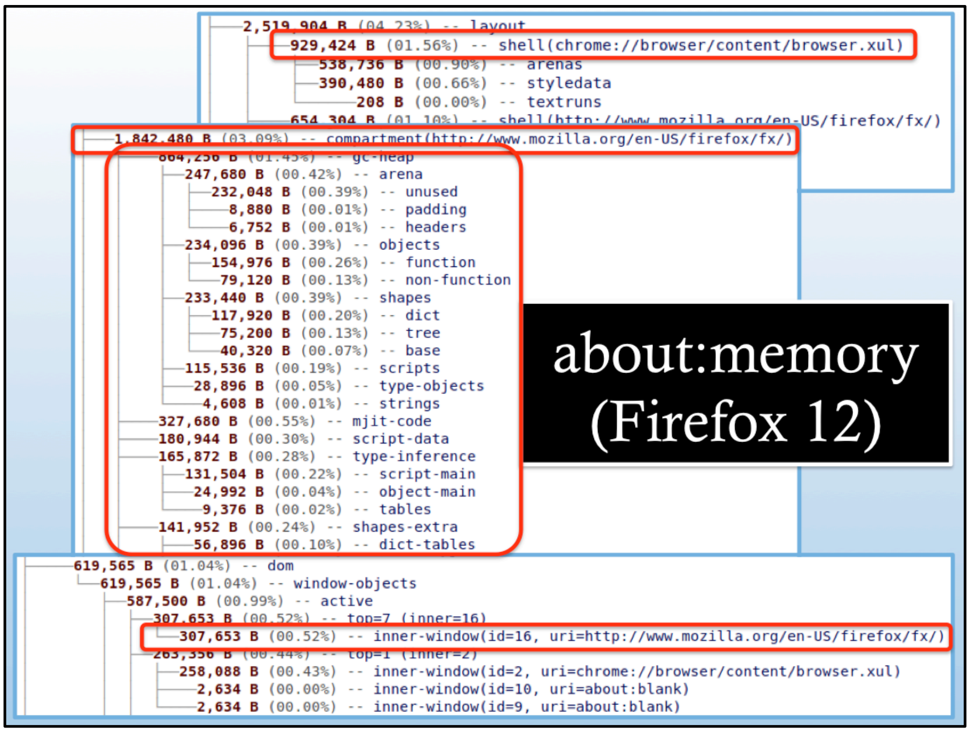
- The whole page is very minimalist, and there are two reasons for this.
- First, if it's too fancy it'll perturb the browser's memory consumption a lot.
- But more importantly, I very much wanted users to be able to cut and paste its contents into bug reports, because that's so much easier than taking screenshots. The old about:memory was terrible for this. Even though it was all text, if you cut and paste it into a text buffer all the numbers were jammed up against the labels and you had to reformat it by hand to make it readable.
- So in the new about:memory, the lines indicating the tree structure are just Unicode box-drawing characters, and all the HTML elements have just the right whitespace between them. As a result, it reproduces beautifully when you cut and paste, and this has been really useful.



about:memory (Firefox 12)

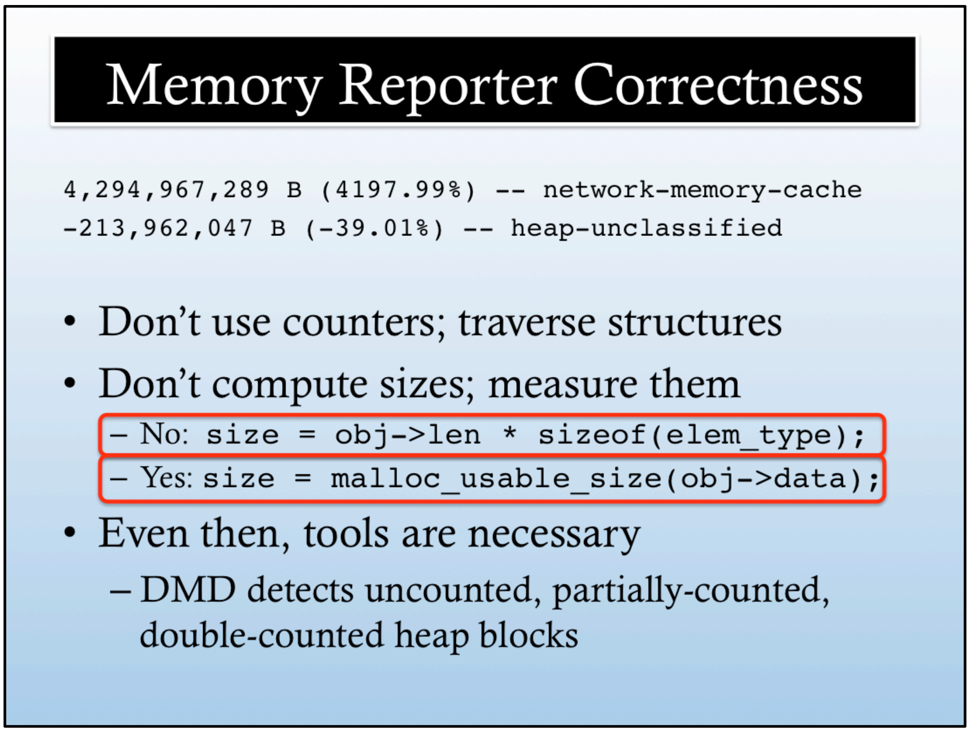
```
──2,519,904 B (04.23%) -- layout
  ├──929,424 B (01.56%) -- shell(chrome://browser/content/browser.xul)
  │  ├──538,736 B (00.90%) -- arenas
  │  ├──390,480 B (00.66%) -- styledata
  │  └──────208 B (00.00%) -- textruns
  └──654,304 B (01.10%) -- shell(http://www.mozilla.org/en-US/firefox/fx/)
──1,842,480 B (03.09%) -- compartment(http://www.mozilla.org/en-US/firefox/fx/)
  ├──864,256 B (01.45%) -- gc-heap
  │  ├──247,680 B (00.42%) -- arena
  │  │  ├──232,048 B (00.39%) -- unused
  │  │  ├────8,880 B (00.01%) -- padding
  │  │  └────6,752 B (00.01%) -- headers
  │  ├──234,096 B (00.39%) -- objects
  │  │  ├──154,976 B (00.26%) -- function
  │  │  └───79,120 B (00.13%) -- non-function
  │  ├──233,440 B (00.39%) -- shapes
  │  │  ├──117,920 B (00.20%) -- dict
  │  │  ├───75,200 B (00.13%) -- tree
  │  │  └───40,320 B (00.07%) -- base
  │  ├──115,536 B (00.19%) -- scripts
  │  ├───28,896 B (00.05%) -- type-objects
  │  └────4,608 B (00.01%) -- strings
  ├──327,680 B (00.55%) -- mjit-code
  ├──180,944 B (00.30%) -- script-data
  ├──165,872 B (00.28%) -- type-inference
  │  ├──131,504 B (00.22%) -- script-main
  │  ├───24,992 B (00.04%) -- object-main
  │  └────9,376 B (00.02%) -- tables
  ├──141,952 B (00.24%) -- shapes-extra
  │  ├───56,896 B (00.10%) -- dict-tables
──619,565 B (01.04%) -- dom
  └──619,565 B (01.04%) -- window-objects
     ├──587,500 B (00.99%) -- active
     │  ├──307,653 B (00.52%) -- top=7 (inner=16)
     │  │  └──307,653 B (00.52%) -- inner-window(id=16, uri=http://www.mozilla.org/en-US/firefox/fx/)
     │  ├──263,356 B (00.44%) -- top=1 (inner=2)
     │  │  ├──258,088 B (00.43%) -- inner-window(id=2, uri=chrome://browser/content/browser.xul)
     │  │  ├────2,634 B (00.00%) -- inner-window(id=10, uri=about:blank)
     │  │  └────2,634 B (00.00%) -- inner-window(id=9, uri=about:blank)
```

- So all that was a big improvement, but about:memory didn't become a really useful tool until Firefox 7, when I split up the reporting of the JavaScript engine's memory usage into compartments.
- A compartment is a data structure that holds the JavaScript data for all the scripts that belong to a single domain. So this more or less tells you how much JavaScript memory is being used by each website. And that's a killer feature.
- First of all, it became obvious that sometimes we were creating many more compartments than necessary. If you opened up TechCrunch.com you'd get 70 compartments. Once we realized this, we soon managed to get it down to 8.
- More importantly, as soon as this feature was implemented people started reporting that they had compartments alive for websites they'd closed hours ago. We call these zombie compartments, and they are bad memory leaks.
- We quickly found and fixed several zombie compartments in Firefox. People still get them now, but they're almost always caused by add-ons, and that's a topic I'll return to shortly.



```
──2,519,904 B (04.23%) -- layout
  ├──929,424 B (01.56%) -- shell(chrome://browser/content/browser.xul)
  │  ├──538,736 B (00.90%) -- arenas
  │  ├──390,480 B (00.66%) -- styledata
  │  └──────208 B (00.00%) -- textruns
  ├──654,304 B (01.10%) -- shell(http://www.mozilla.org/en-US/firefox/fx/)
──1,842,480 B (03.09%) -- compartment(http://www.mozilla.org/en-US/firefox/fx/)
  ├──864,256 B (01.45%) -- gc-heap
  │  ├──247,680 B (00.42%) -- arena
  │  │  ├──232,048 B (00.39%) -- unused
  │  │  ├────8,880 B (00.01%) -- padding
  │  │  └────6,752 B (00.01%) -- headers
  │  ├──234,096 B (00.39%) -- objects
  │  │  ├──154,976 B (00.26%) -- function
  │  │  └───79,120 B (00.13%) -- non-function
  │  ├──233,440 B (00.39%) -- shapes
  │  │  ├──117,920 B (00.20%) -- dict
  │  │  ├───75,200 B (00.13%) -- tree
  │  │  └───40,320 B (00.07%) -- base
  │  ├──115,536 B (00.19%) -- scripts
  │  ├───28,896 B (00.05%) -- type-objects
  │  └────4,608 B (00.01%) -- strings
  ├──327,680 B (00.55%) -- mjit-code
  ├──180,944 B (00.30%) -- script-data
  ├──165,872 B (00.28%) -- type-inference
  │  ├──131,504 B (00.22%) -- script-main
  │  ├───24,992 B (00.04%) -- object-main
  │  └────9,376 B (00.02%) -- tables
  ├──141,952 B (00.24%) -- shapes-extra
  │  ├───56,896 B (00.10%) -- dict-tables
──619,565 B (01.04%) -- dom
  └──619,565 B (01.04%) -- window-objects
     ├──587,500 B (00.99%) -- active
     │  ├──307,653 B (00.52%) -- top=7 (inner=16)
     │  │  └──307,653 B (00.52%) -- inner-window(id=16, uri=http://www.mozilla.org/en-US/firefox/fx/)
     │  ├──263,356 B (00.44%) -- top=1 (inner=2)
     │  │  ├──258,088 B (00.43%) -- inner-window(id=2, uri=chrome://browser/content/browser.xul)
     │  │  ├────2,634 B (00.00%) -- inner-window(id=10, uri=about:blank)
     │  │  └────2,634 B (00.00%) -- inner-window(id=9, uri=about:blank)
```

- We now have similar break-downs for layout memory and DOM memory.
- And we're working towards unifying these so that we can provide something that users have been requesting for years, which is to report how much memory each tab is using. This will be a really nice feature, because if the browser is using a lot of memory, you can see which tabs to close.
- We've also broken down the memory usage into more detailed categories, which is helpful to Firefox developers. For example, there are now something like 24 different measurements that are made for each JavaScript compartment.

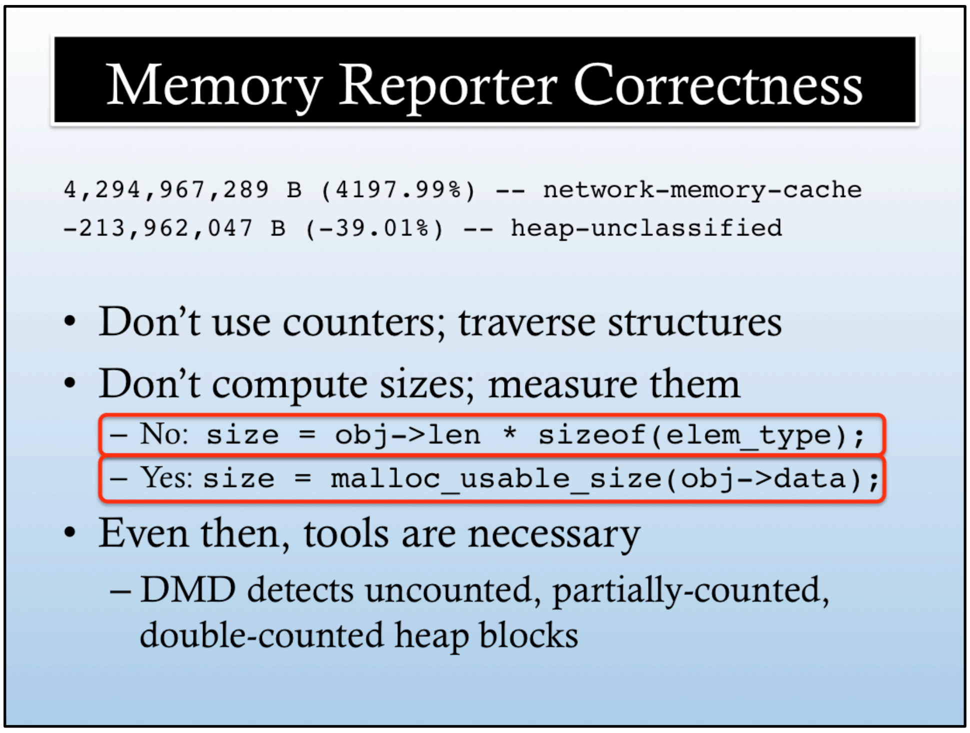


Memory Reporter Correctness

```
4,294,967,289 B (4197.99%) -- network-memory-cache
-213,962,047 B (-39.01%) -- heap-unclassified
```

- Don’t use counters; traverse structures
- Don’t compute sizes; measure them
  - No: `size = obj->len * sizeof(elem_type);`
  - Yes: `size = malloc_usable_size(obj->data);`
- Even then, tools are necessary
  - DMD detects uncounted, partially-counted, double-counted heap blocks

- So about:memory has become a really powerful tool. But how do you know its numbers are reliable? The implementation involves things called memory reporters, and each one reports one or more numbers.
- And it turns out it’s very easy for memory reporters to be buggy, but it can be very difficult to know if that’s the case. Sometimes you get something ridiculous like a percentage greater than 100, or a negative number, but bugs aren’t always that obvious. And about:memory is horribly non-deterministic – you get different numbers every time you run it -- which means it’s really hard to write automated tests for it.
- So we’ve developed a couple of good practices to follow when writing memory reporters.
- If you have a large data structure, one way to measure its size is to maintain a counter, and update that counter every time the data structure grows or shrinks. It turns out this is very error-prone, because when you modify the data structure’s code it’s easy to forget to update the code for the counter. We’ve found that it’s less error-prone to instead traverse the data structure each time you need a measurement. That’s slower, but that’s ok because memory reporters aren’t performance-critical.
- Another thing we’ve learnt is that when dealing with heap blocks, computing their size analytically is error-prone. It’s much better to just measure them with malloc_usable_size() or a similar function.

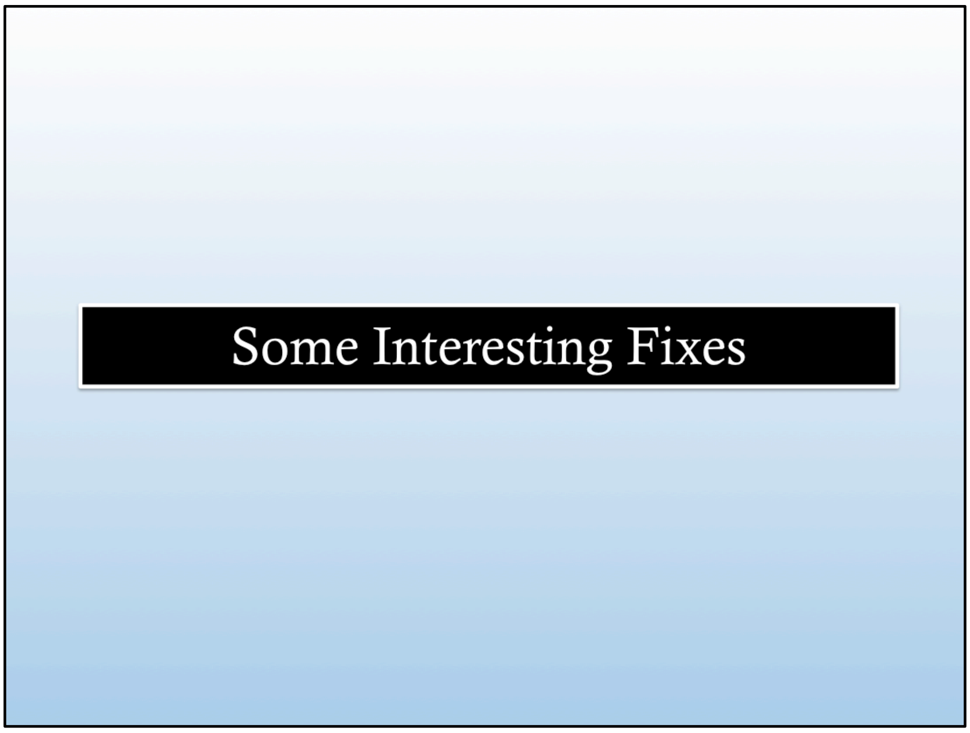


Memory Reporter Correctness

```
4,294,967,289 B (4197.99%) -- network-memory-cache
-213,962,047 B (-39.01%) -- heap-unclassified
```

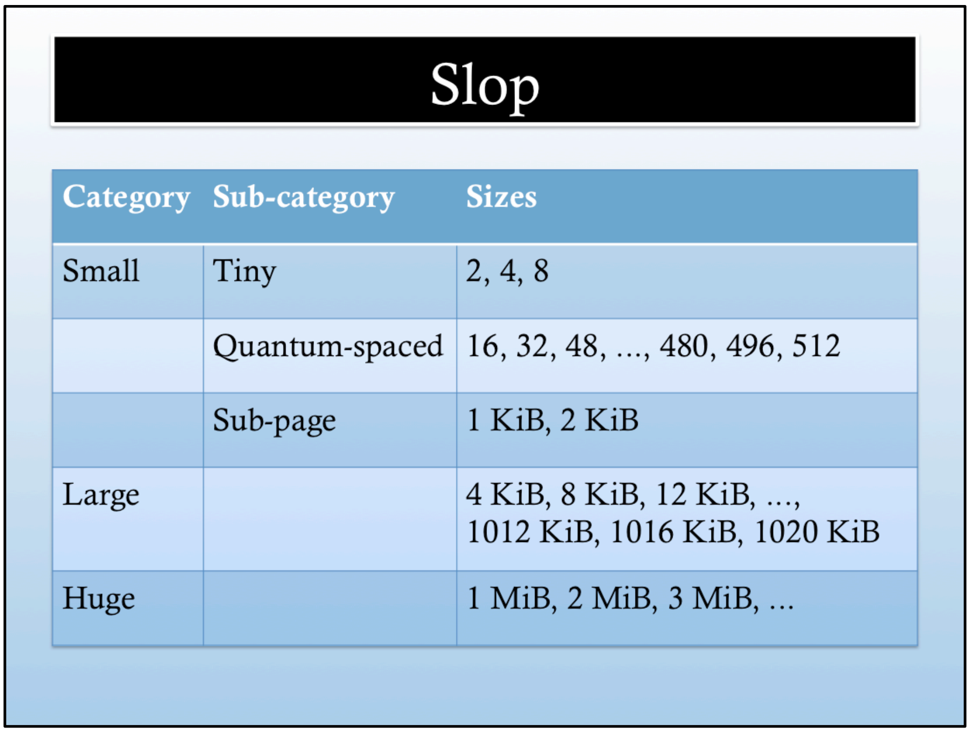
- Don't use counters; traverse structures
- Don't compute sizes; measure them
  - No: `size = obj->len * sizeof(elem_type);`
  - Yes: `size = malloc_usable_size(obj->data);`
- Even then, tools are necessary
  - DMD detects uncounted, partially-counted, double-counted heap blocks

- But finally, even if you follow these guidelines, it's still easy to get things wrong. To really make things reliable requires an automated tool, so I wrote one called DMD (short for "Dark Matter Detector"). It's built with Valgrind, and it tracks every block allocated on the heap. I also added some annotations to Firefox that tell DMD when Firefox has counted a heap block. So DMD can detect blocks that are counted twice, and also blocks that aren't counted at all.
- Thanks to DMD, dark matter is typically around 20%, and I'm hoping to get it below 10% over the next few months.



Some Interesting Fixes

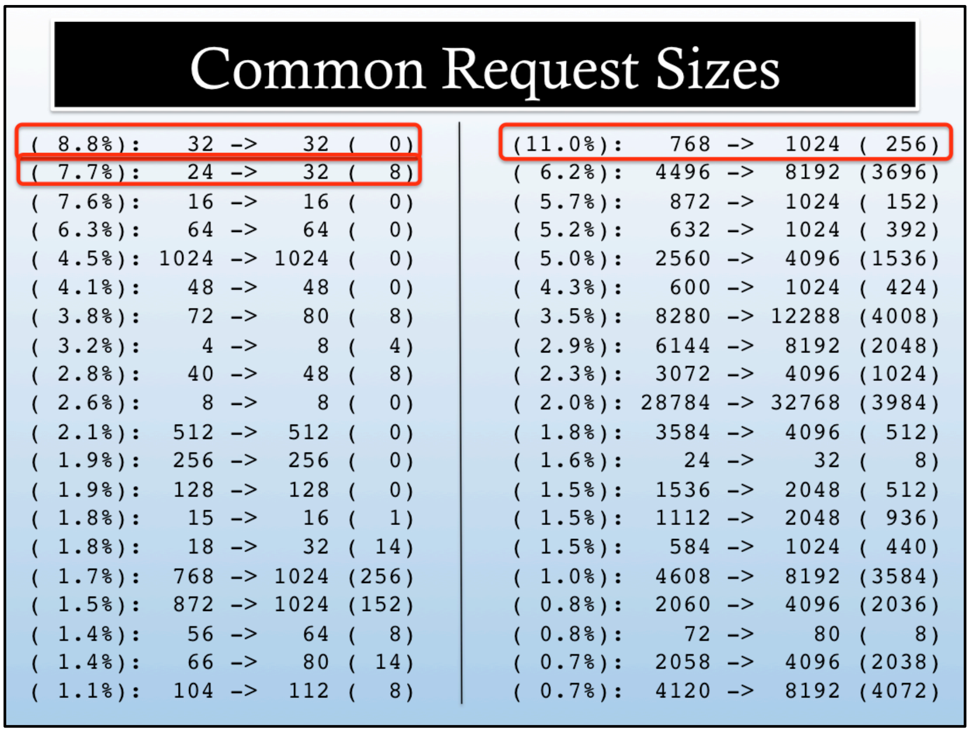
- So that's enough about memory profiling. I'm now going to talk about an interesting class of problems and their fixes.



Slop

Category	Sub-category	Sizes
Small	Tiny	2, 4, 8
	Quantum-spaced	16, 32, 48, …, 480, 496, 512
	Sub-page	1 KiB, 2 KiB
Large		4 KiB, 8 KiB, 12 KiB, …, 1012 KiB, 1016 KiB, 1020 KiB
Huge		1 MiB, 2 MiB, 3 MiB, …

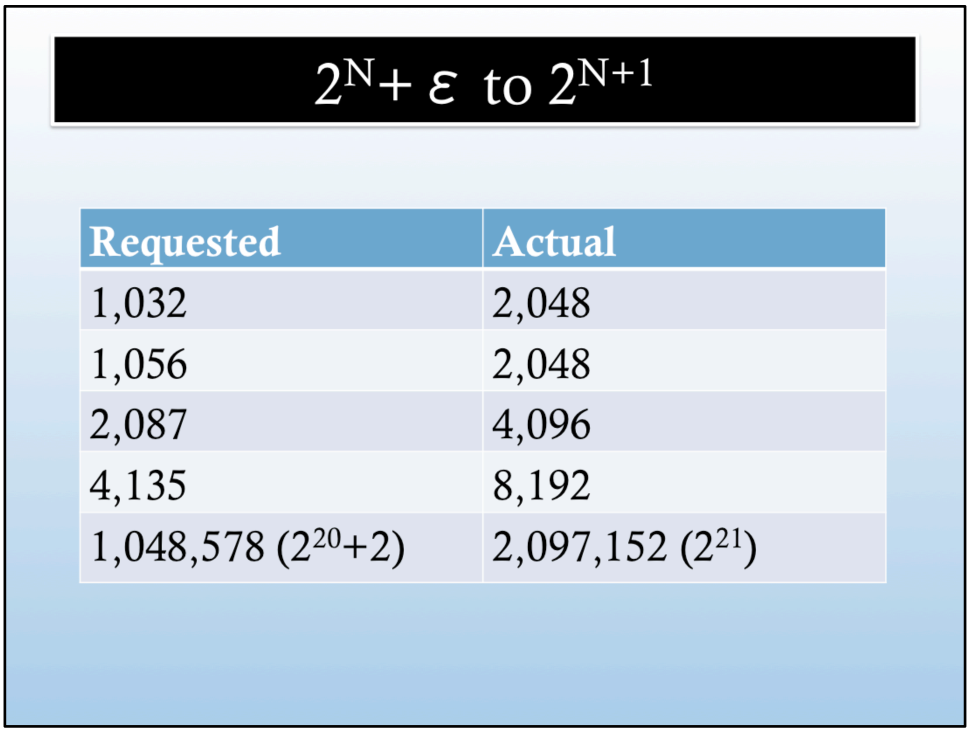
- These fixes all relate to a particular feature of heap allocators, which is that when you request a certain number of bytes with malloc or new, the allocator often gives you more than you asked for. This wastes some memory, but it helps keep allocators fast and simple, and minimizes fragmentation when blocks are freed.
- Firefox uses a heap allocator called jemalloc, and this table indicates its size classes. If you request any number of bytes that isn't one of the numbers in this table, jemalloc will round up the request to the next size class. For example, if you request 500 bytes it'll round up to 512. In quite a few cases you'll get almost twice the amount of memory you asked for: if you ask for 513 bytes, you'll get 1024; if you ask for 4097 you'll get 8192.
- These wasted bytes are often called internal fragmentation, but I prefer the term "slop" because it's much faster to type.
- A lot of the time you just have to live with slop. Many allocations have a fixed size, and for this reason something like 5 to 10% of Firefox's heap memory is slop.
- However, sometimes you have flexibility in your request size. In those cases, it's a good idea to request a size that's a power-of-two, because in jemalloc power-of-two-sized allocations never have any slop.



Common Request Sizes

```
( 8.8%):   32 ->   32 (  0)      (11.0%):   768 ->  1024 ( 256)
( 7.7%):   24 ->   32 (  8)      ( 6.2%):  4496 ->  8192 (3696)
( 7.6%):   16 ->   16 (  0)      ( 5.7%):   872 ->  1024 ( 152)
( 6.3%):   64 ->   64 (  0)      ( 5.2%):   632 ->  1024 ( 392)
( 4.5%): 1024 -> 1024 (  0)      ( 5.0%):  2560 ->  4096 (1536)
( 4.1%):   48 ->   48 (  0)      ( 4.3%):   600 ->  1024 ( 424)
( 3.8%):   72 ->   80 (  8)      ( 3.5%):  8280 -> 12288 (4008)
( 3.2%):    4 ->    8 (  4)      ( 2.9%):  6144 ->  8192 (2048)
( 2.8%):   40 ->   48 (  8)      ( 2.3%):  3072 ->  4096 (1024)
( 2.6%):    8 ->    8 (  0)      ( 2.0%): 28784 -> 32768 (3984)
( 2.1%):  512 ->  512 (  0)      ( 1.8%):  3584 ->  4096 ( 512)
( 1.9%):  256 ->  256 (  0)      ( 1.6%):    24 ->    32 (   8)
( 1.9%):  128 ->  128 (  0)      ( 1.5%):  1536 ->  2048 ( 512)
( 1.8%):   15 ->   16 (  1)      ( 1.5%):  1112 ->  2048 ( 936)
( 1.8%):   18 ->   32 ( 14)      ( 1.5%):   584 ->  1024 ( 440)
( 1.7%):  768 -> 1024 (256)      ( 1.0%):  4608 ->  8192 (3584)
( 1.5%):  872 -> 1024 (152)      ( 0.8%):  2060 ->  4096 (2036)
( 1.4%):   56 ->   64 (  8)      ( 0.8%):    72 ->    80 (   8)
( 1.4%):   66 ->   80 ( 14)      ( 0.7%):  2058 ->  4096 (2038)
( 1.1%):  104 ->  112 (  8)      ( 0.7%):  4120 ->  8192 (4072)
```

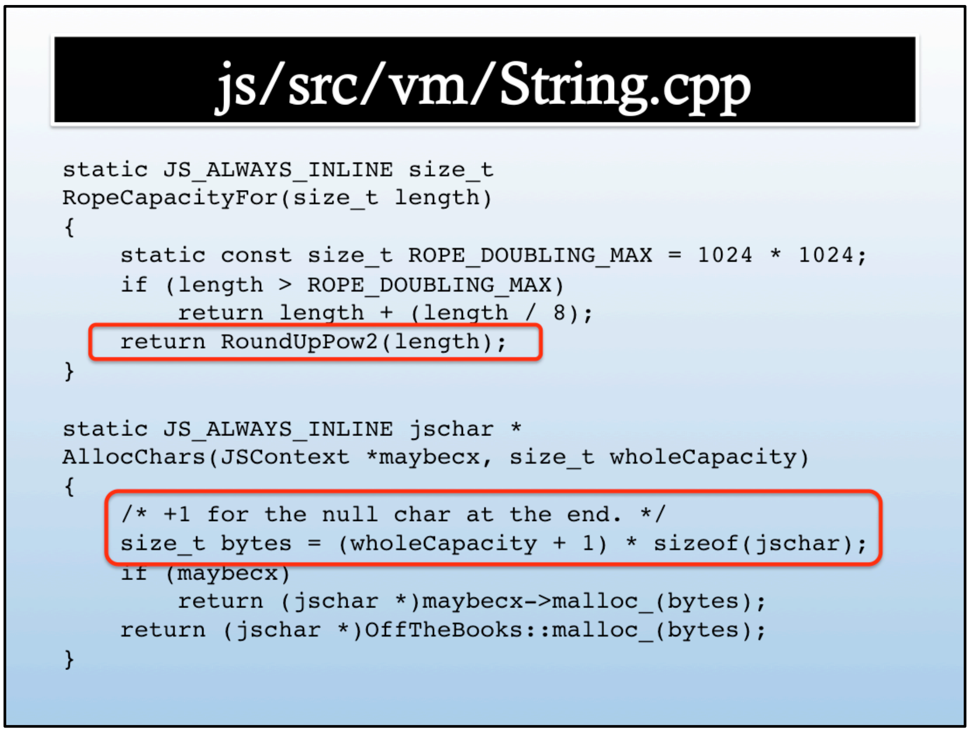
- One day I measured how much slop we had. I instrumented jemalloc to print, for every allocation, the request size, the rounded up size, and the difference between these two, which is the amount of slop. I ran Firefox to get some data and then ran that data through a concordance script I have that tells me how many times each unique case occurs.
- I redid this just last week and got the following data. On the LHS of this slide are the 20 most frequent cases. Many of them involve no slop, for example, the most common request size is 32 bytes, which accounts for 8.8% of all allocations. But the second-most common request size is 24, which gets rounded up to 32.
- On the RHS is the 20 most frequent cases once they are weighted by the amount of slop. The most important case is allocations of 768 bytes, which get rounded up to 1024 bytes, wasting 256; that accounts for 11% of the total slop allocated.



$2^N + \varepsilon$ to 2^{N+1}

Requested	Actual
1,032	2,048
1,056	2,048
2,087	4,096
4,135	8,192
1,048,578 (2^{20}+2)	2,097,152 (2^{21})

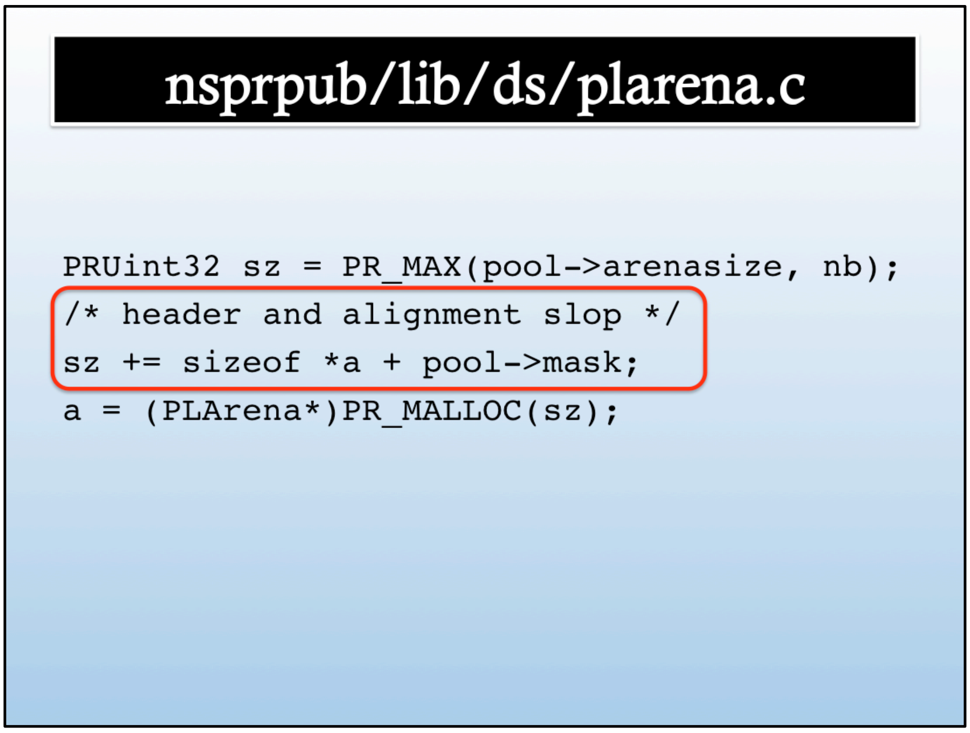
- So that's what it looks like now, but when I ran this in August last year it looked a bit different. In particular, there were an awful lot of cases where a number slightly larger than a power-of-two was rounded up to the next power-of-two. For example, requests for 1032 bytes were rounded up to 2048 bytes; requests for 2087 bytes were rounded up to 4096, and requests for 1MB plus 2 bytes were rounded up to 2MB. It was this last one that first caught my attention and got me thinking about this.
- What I found was four distinct places in the codebase where the code in question had flexibility in the amount of memory it allocated, and had intended to ask for a power-of-two, but had botched the size computation and thus ended up asking for slightly more than a power-of-two!



js/src/vm/String.cpp

```
static JS_ALWAYS_INLINE size_t
RopeCapacityFor(size_t length)
{
    static const size_t ROPE_DOUBLING_MAX = 1024 * 1024;
    if (length > ROPE_DOUBLING_MAX)
        return length + (length / 8);
    return RoundUpPow2(length);
}

static JS_ALWAYS_INLINE jschar *
AllocChars(JSContext *maybecx, size_t wholeCapacity)
{
    /* +1 for the null char at the end. */
    size_t bytes = (wholeCapacity + 1) * sizeof(jschar);
    if (maybecx)
        return (jschar *)maybecx->malloc_(bytes);
    return (jschar *)OffTheBooks::malloc_(bytes);
}
```

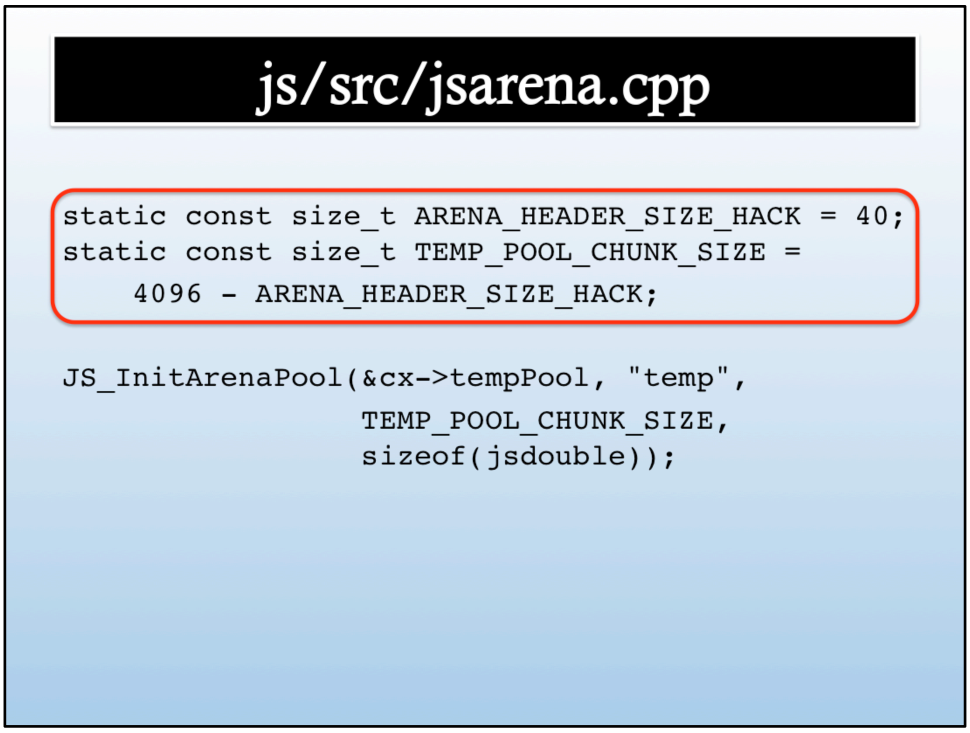
- The first case involved JavaScript strings.
- When we concatenate two JavaScript strings, we over-allocate the buffer of the resulting string, which is reasonable because if it's appended to later, we might be able to add the extra characters inline and avoid another allocation.
- But the code first rounded up the size to a power-of-two…
- …and then later on added 1 for the terminating NUL character. Whoops.



nsprpub/lib/ds/plarena.c

```
PRUint32 sz = PR_MAX(pool->arenasize, nb);
/* header and alignment slop */
sz += sizeof *a + pool->mask;
a = (PLArena*)PR_MALLOC(sz);
```

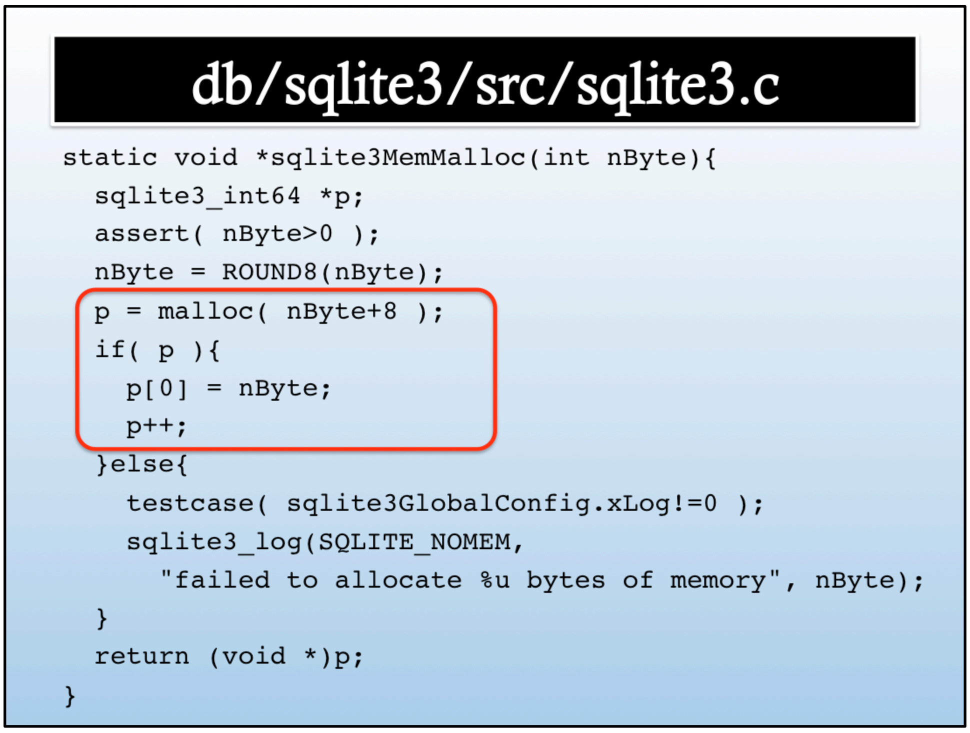
- The second case involved PLArenaPool, which is the arena allocator I mentioned earlier. Each arena it allocates has space for data, and that space has a size that is a power-of-two, most commonly 4KB. But then each arena also has a header containing a few words of book-keeping data, which caused the allocation to be rounded up to to 8KB. In other words, almost half the memory allocated by PLArenaPool was wasted.
- This allocator is used a lot for doing page layout, and when I fixed this it avoided about 3MB of wasted space just for Gmail. We even found one case, which was a gigantic page of text, where this was causing 700MB of wasted memory, and fixing the problem reduced Firefox's memory consumption for that page from 2GB to 1.3GB. And tragically enough, a bug report had been filed for this exact problem 3.5 years earlier, but it was never fixed.



js/src/jsarena.cpp

```
static const size_t ARENA_HEADER_SIZE_HACK = 40;
static const size_t TEMP_POOL_CHUNK_SIZE =
    4096 - ARENA_HEADER_SIZE_HACK;

JS_InitArenaPool(&cx->tempPool, "temp",
                 TEMP_POOL_CHUNK_SIZE,
                 sizeof(jsdouble));
```

- The third case involved JSArenaPool, which is a copy of the PLArenaPool code that was modified for use in the JavaScript engine. This code had exactly the same problem. Now, it turns out this wasn't wasting much memory in practice because someone had realized this problem existed, and for the most widely used arena pool they'd worked around it by requesting arenas that were slightly less than 4KB. They'd defined ARENA_HEADER_SIZE_HACK = 40, and TEMP_POOL_CHUNK_SIZE = 4096 – ARENA_HEADER_SIZE_HACK. But there was no comment describing how this worked, and they hadn't thought to fix this problem more generally.

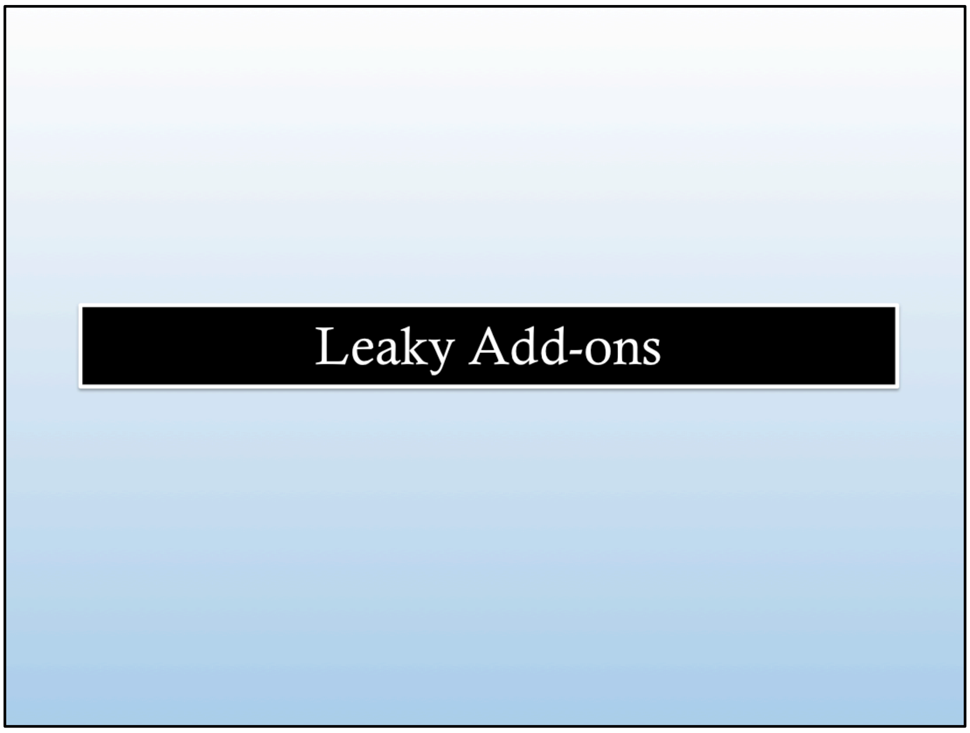


db/sqlite3/src/sqlite3.c

```
static void *sqlite3MemMalloc(int nByte){
  sqlite3_int64 *p;
  assert( nByte>0 );
  nByte = ROUND8(nByte);
  p = malloc( nByte+8 );
  if( p ){
    p[0] = nByte;
    p++;
  }else{
    testcase( sqlite3GlobalConfig.xLog!=0 );
    sqlite3_log(SQLITE_NOMEM,
      "failed to allocate %u bytes of memory", nByte);
  }
  return (void *)p;
}
```

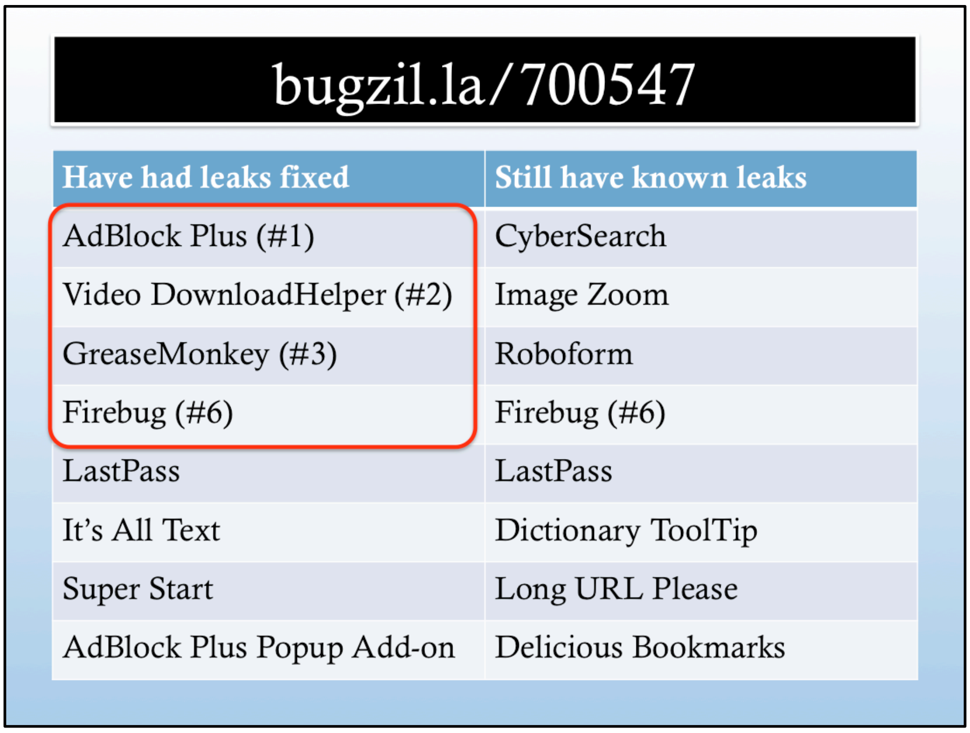
- The fourth case involved SQLite, which is a database implementation embedded in Firefox.
- SQLite is very careful about measuring its memory usage and provides an API to access those measurements.
- But in order to measure its own memory usage accurately, it adds 8 bytes to every allocation request and stores the requested size in those extra 8 bytes.
- A lot of SQLite's allocation requests have sizes that are a power-of-two, and so these extra 8 bytes were causing the allocations to double in size. So, ironically enough, the slop caused by storing the requested size caued that size to be inaccurate. As a result, SQLite was significantly under-reporting its own memory consumption.

- Those were the four cases I found to begin with. I subsequently found and fixed a few more, and Firefox is now nearly free of such cases.
- So what's with the picture? I have a colleague named Andreas Gal, and when he sees really badly written code, he uses the word "clownshoes" to describe it.
- When I found the first of these bugs, I was so amused yet horrified that I put a "clownshoes" tag on the bug report. And then later on I realized that not only are these allocations comically bad, but they're like a pair of clownshoes in that they're much bigger than they need to be because they contain a lot of empty space.
- So the name has stuck, and we now call these clownshoes allocations.



Leaky Add-ons

- Another big class of problems involves add-ons.
- Add-ons are a critical feature of Firefox. They are arguably the single feature that most differentiates Firefox from other browsers.
- Firefox add-ons can modify just about anything about the browser. This makes them very powerful and flexible. But they can also cause huge problems if they are badly written, and many of them are.



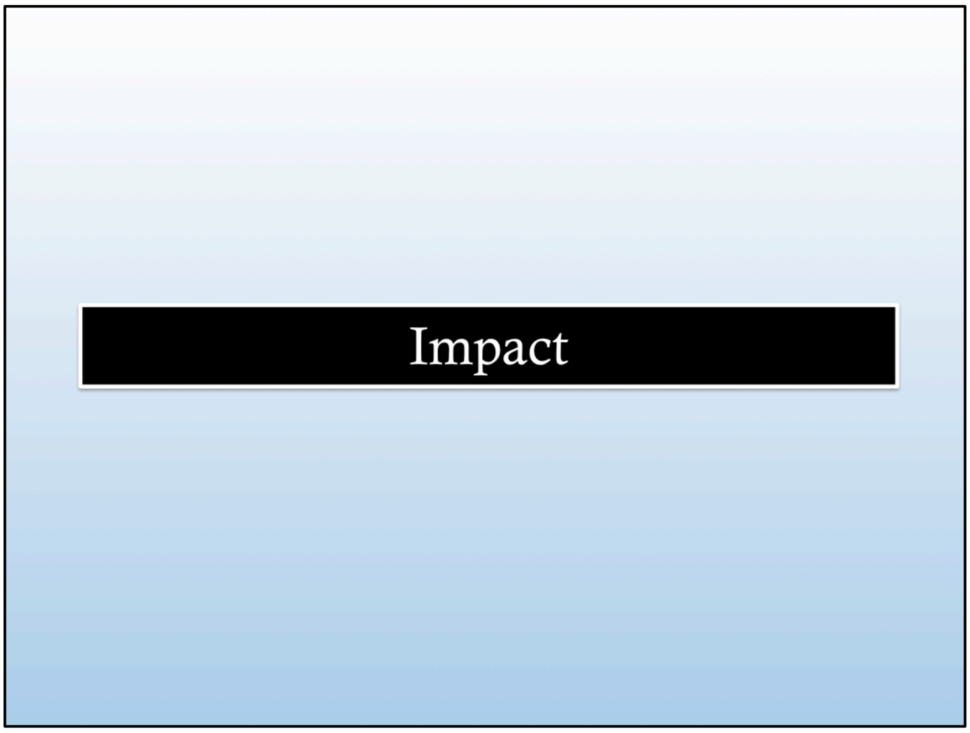
bugzil.la/700547

Have had leaks fixed	Still have known leaks
AdBlock Plus (#1)	CyberSearch
Video DownloadHelper (#2)	Image Zoom
GreaseMonkey (#3)	Roboform
Firebug (#6)	Firebug (#6)
LastPass	LastPass
It's All Text	Dictionary ToolTip
Super Start	Long URL Please
AdBlock Plus Popup Add-on	Delicious Bookmarks

- In particular, many of them have memory leaks. This was suspected for a long time, but until we had per-compartment reporting in about:memory we didn't really have any tools to demonstrate that conclusively. In the six months since that feature was added, we've discovered quite a few add-ons that create zombie compartments. This slide shows some of them.
- Notice that the first four add-ons listed in the left-hand column – AdBlock Plus, Video DownloadHelper, GreaseMonkey, and Firebug – are among the top 10 most popular add-ons that are available from Mozilla's add-on site. Given that the popular add-ons tend to have higher-than-average code quality, we strongly suspect that this list is just the tip of the iceberg, that a sizeable fraction of all add-ons have memory leaks.
- In fact, these days, when we get a bug report where someone is complaining about excessive memory consumption, it's usually an add-on that's at fault.
- A lot of these leaks have the same cause: an add-on stores a reference to a window or a DOM node and then fails to release that reference when the window closes or the page unloads.

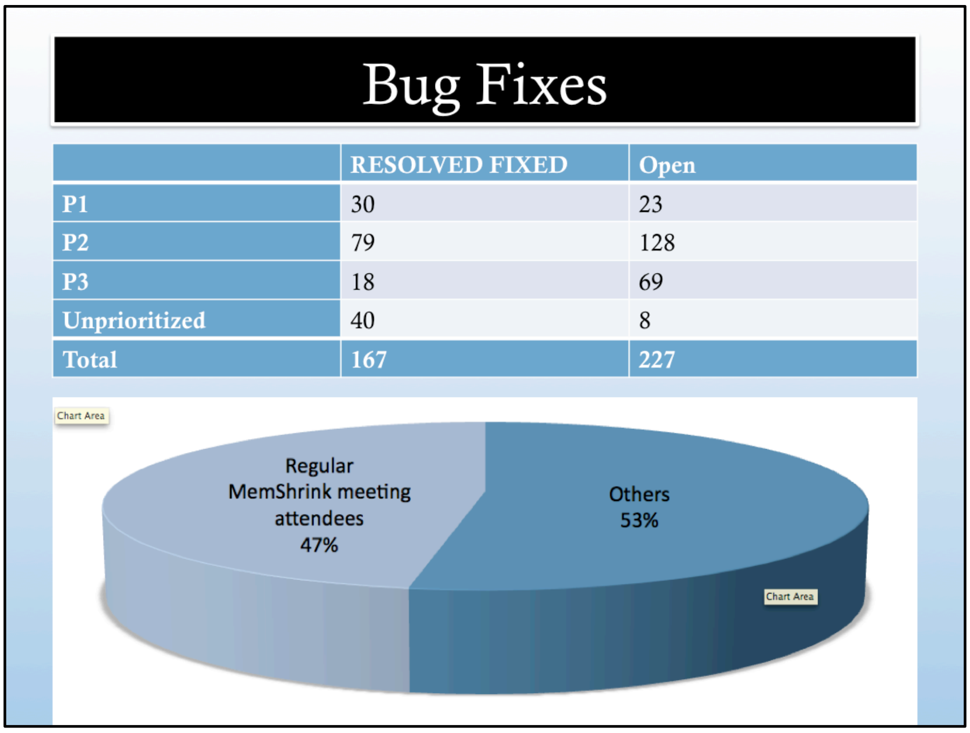
s/Firefox/Scapegoat/

- This is bad. Although these leaks are not Mozilla's fault, they are Mozilla's problem. Many Firefox users have add-ons installed -- some people have 20 or 30 or more -- and Firefox gets blamed for the sins of its add-ons.
- But there is some hope. Two members of the add-ons team came to last week's MemShrink meeting, and we have two simple ideas to help.
- First, we want to add a feature to Mozilla's add-ons site so that any add-on which is known to leak memory can be marked as such. In fact, multiple flags to indicate various performance problems would be useful. This would mean that users who are installing that add-on have some idea that it might cause problems.
- Second, we want to tighten the review procedure. Each new add-on and each new version of an existing add-on must pass a review before it gets put onto the add-ons site. We want to add an item to the reviewer's checklist, which is to test for zombie compartments. Many of the leaks we've found in add-ons would have been caught with only a couple of minutes of testing.
- So this is a big problem, but hopefully it'll start to get better soon.



Impact

- At this point, I've talked about some of the work we've done for MemShrink.
- You might be wondering how much impact this work has had. The short answer is "quite a lot".



Bug Fixes

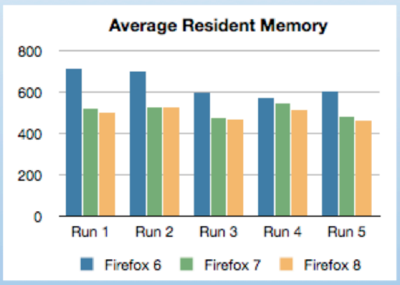
	RESOLVED FIXED	Open
P1	30	23
P2	79	128
P3	18	69
Unprioritized	40	8
Total	167	227

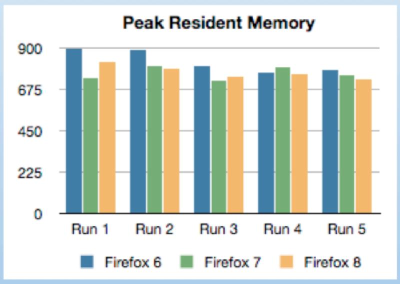
Chart Area
Regular MemShrink meeting attendees 47%
Others 53%
Chart Area

- Let’s look first at the raw bug counts. In the six months since MemShrink started in earnest, 167 bugs with the MemShrink tag have been resolved as FIXED, and that excludes bugs that have been closed because they were invalid, incomplete or duplicates.
- There’s a core group of 6 people who regularly attend MemShrink meetings, and roughly half the bugs have been fixed by that group. The other half of the bug fixes have been spread across more 50 people. I think this is a good spread.
- But there are still 227 bugs open, so there’s still plenty of improvements to be made. But one nice thing is that, unlike early on, the vast majority of these are not problems reported by users, but ideas from Firefox developers for things that can be improved.

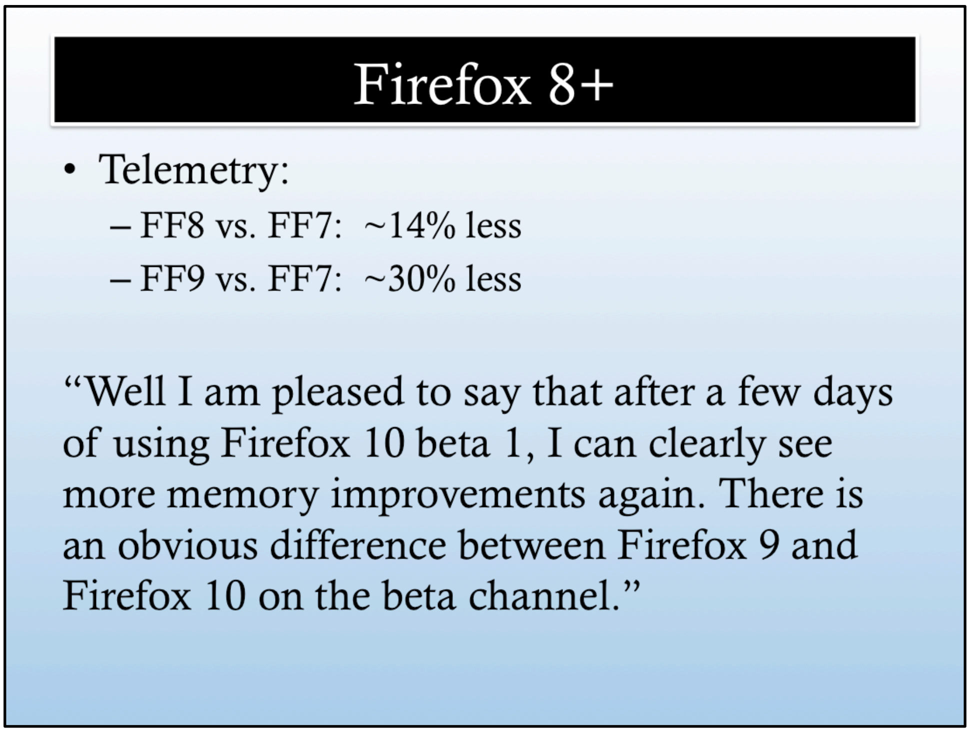
Firefox 7

- “Firefox 7 uses less memory than Firefox 6 (and 5 and 4): often 20% to 30% less, and sometimes as much as 50% less… This means that Firefox 7 is faster (sometimes drastically so) and less likely to crash”





- But what does this mean for Firefox in practice?
- Firefox 7 was the version under development when MemShrink started. Enough memory consumption improvements were made that they became the primary focus of the publicity for that release.
- As part of that publicity, I wrote a blog post where I summarized the improvements as follows: …
- Those measurements were for physical memory consumption. I was initially very reluctant to put hard numbers in this post because the results varied so much depending on workload. But the marketing people encouraged me to, and I need not have worried. Hardly anyone tried to verify the numbers, and those few that did got similar results. Most tech press outlets just quoted me verbatim, which was quite disappointing.

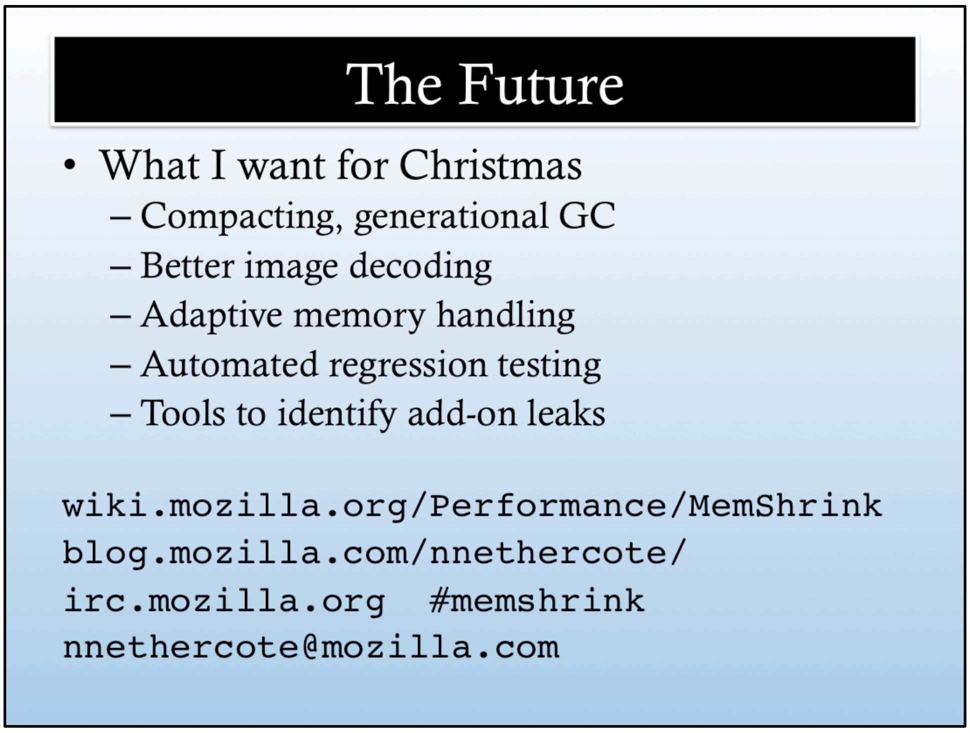


Firefox 8+

- Telemetry:
  - FF8 vs. FF7: ~14% less
  - FF9 vs. FF7: ~30% less

"Well I am pleased to say that after a few days of using Firefox 10 beta 1, I can clearly see more memory improvements again. There is an obvious difference between Firefox 9 and Firefox 10 on the beta channel."

- As for the versions since then, one member of the Mozilla metrics team is using telemetry data to analyze the of memory consumption of Firefox 7, 8 and 9. He hasn't finished it yet but his preliminary results show that Firefox 8's average physical memory consumption is 14% lower than Firefox 7's, and Firefox 9's is 30% lower than Firefox 7's.
- Beyond that, we're currently in the middle of the Firefox 12 development cycle, and I'm confident the downward trend has continued, and we have had lots of positive comments from users.
- But we don't have much in the way of numbers. We do have some automatic regression testing, but it's not very good, and we have people are working on improving this.

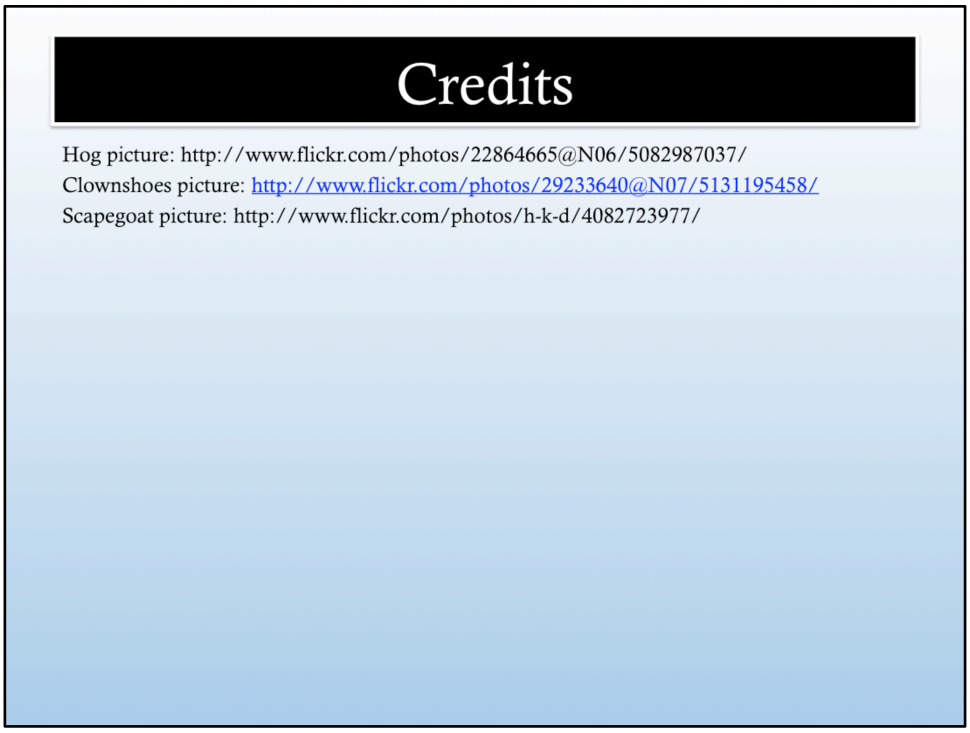


The Future

- What I want for Christmas
  - Compacting, generational GC
  - Better image decoding
  - Adaptive memory handling
  - Automated regression testing
  - Tools to identify add-on leaks

```
wiki.mozilla.org/Performance/MemShrink
blog.mozilla.com/nnethercote/
irc.mozilla.org  #memshrink
nnethercote@mozilla.com
```

- So I'm confident that MemShrink has been a success so far, but there's plenty of work still to do and I don't see it winding down for some time yet.
- Looking forward, there are a number of big things I'd like to see.
- First is a compacting, generational garbage collector for the JavaScript engine. There's currently a lot of fragmentation in the JavaScript heap, and this would reduce that drastically. There are a couple of people working actively on this, which is great.
- Second is better image decoding. When you open a page in the foreground tab, we immediately decompress all the images within it and don't discard any of that data until we switch to a different tab. On image-heavy pages this sends memory consumption through the roof, and other browsers do a much better job. I don't think anyone is really working on this, sadly.
- Third is adaptive memory handling. On machines with little memory we can do a better job of detecting when memory is low and discarding unnecessary data.
- Fourth is automated regression testing. This will give us a better idea of the progress we're making, and help us avoid going backwards.
- Fifth is tools to identify add-on leaks. about:memory can tell you that an add-on causes a zombie compartment, but it cannot tell you why that compartment is still alive. A tool that did this would make things much easier for add-on authors.
- Finally, if you're interested in reading more or helping, please check out the MemShrink wiki page or my blog, or contact me directly.



Credits

Hog picture: http://www.flickr.com/photos/22864665@N06/5082987037/
Clownshoes picture: http://www.flickr.com/photos/29233640@N07/5131195458/
Scapegoat picture: http://www.flickr.com/photos/h-k-d/4082723977/