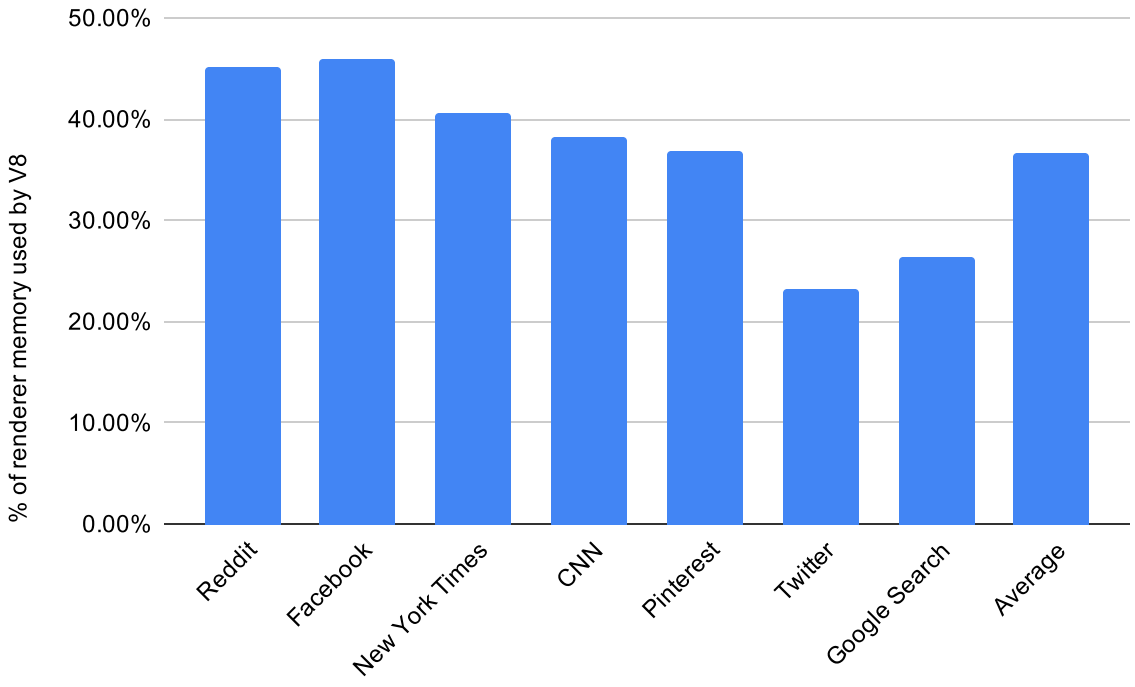
Pointer Compression in V8

Published 30 March 2020 · Tagged with internals memory

There is a constant battle between memory and performance. As users, we would like things to be fast as well as consume as little memory as possible. Unfortunately, usually improving performance comes at a cost of memory consumption (and vice versa).

Back in 2014 Chrome switched from being a 32-bit process to a 64-bit process. This gave Chrome better security, stability and performance, but it came at a memory cost since each pointer now occupies eight bytes instead of four. We took on the challenge to reduce this overhead in V8 to try and get back as many wasted 4 bytes as possible.

Before diving into the implementation, we need to know where we are standing to correctly assess the situation. To measure our memory and performance we use a set of web pages that reflect popular real-world websites. The data showed that V8 contributes up to 60% of Chrome's renderer process memory consumption on desktop, with an average of 40%.



V8 memory consumption percentage in Chrome's renderer memory

Pointer Compression is one of several ongoing efforts in V8 to reduce memory consumption. The idea is very simple: instead of storing 64-bit pointers we can store 32-bit offsets from some "base" address. With such a simple idea, how much can we gain from such a compression in V8?

The V8 heap contains a whole slew of items, such as floating point values, string characters, interpreter bytecode, and tagged values (see next section for details). Upon inspection of the heap, we discovered that on real-world websites these tagged values occupy around 70% of the V8 heap!

Let's take a closer look at what tagged values are.

Value tagging in V8

JavaScript values in V8 are represented as objects and allocated on the V8 heap, no matter if they are objects, arrays, numbers or strings. This allows us to represent any value as a pointer to an object.

Many JavaScript programs perform calculations on integer values, such as incrementing an index in a loop. To avoid us having to allocate a new number object each time an integer is incremented, V8 uses the well-known pointer tagging technique to store additional or alternative data in V8 heap pointers.

The tag bits serve a dual purpose: they signal either strong/weak pointers to objects located in V8 heap, or a small integer. Hence, the value of an integer can be stored directly in the tagged value, without having to allocate additional storage for it.

V8 always allocates objects in the heap at word-aligned addresses, which allows it to use the 2 (or 3, depending on the machine word size) least significant bits for tagging. On 32-bit architectures, V8 uses the least significant bit to distinguish Smis from heap object pointers. For heap pointers, it uses the second least significant bit to distinguish strong references from weak ones:

```
                        |----- 32 bits -----|
Pointer:                |_____address_____w1|
Smi:                    |___int31_value____0|
```

where *w* is a bit used for distinguishing strong pointers from the weak ones.

Note that a Smi value can only carry a 31-bit payload, including the sign bit. In the case of pointers, we have 30 bits that can be used as a heap object address payload. Due to word alignment, the allocation granularity is 4 bytes, which gives us 4 GB of addressable space.

On 64-bit architectures V8 values look like this:

```
            |----- 32 bits -----|----- 32 bits -----|
Pointer:    |________________address_____________w1|
```

```
Smi:        |____int32_value____|00000000000000000|
```

You may notice that unlike 32-bit architectures, on 64-bit architectures V8 can use 32 bits for the Smi value payload. The implications of 32-bit Smis on pointer compression are discussed in the following sections.

Compressed tagged values and new heap layout

With Pointer Compression, our goal is to somehow fit both kinds of tagged values into 32 bits on 64-bit architectures. We can fit pointers into 32 bits by:

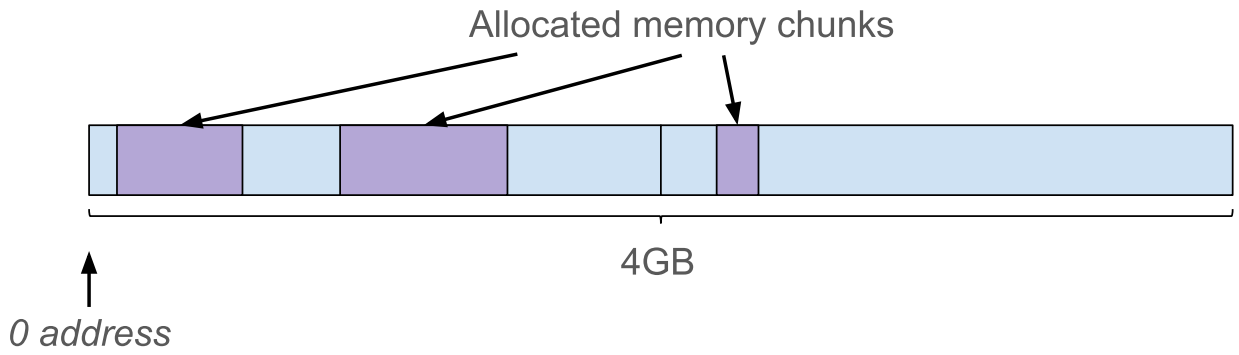
- making sure all V8 objects are allocated within a 4-GB memory range
- representing pointers as offsets within this range

Having such a hard limit is unfortunate, but V8 in Chrome already has a 2-GB or 4-GB limit on the size of the V8 heap (depending on how powerful the underlying device is), even on 64-bit architectures. Other V8 embedders, such as Node.js, may require bigger heaps. If we impose a maximum of 4 GB, it would mean that these embedders cannot use Pointer Compression.

The question is now how to update the heap layout to ensure that 32-bit pointers uniquely identify V8 objects.

Trivial heap layout

The trivial compression scheme would be to allocate objects in the first 4 GB of address space.

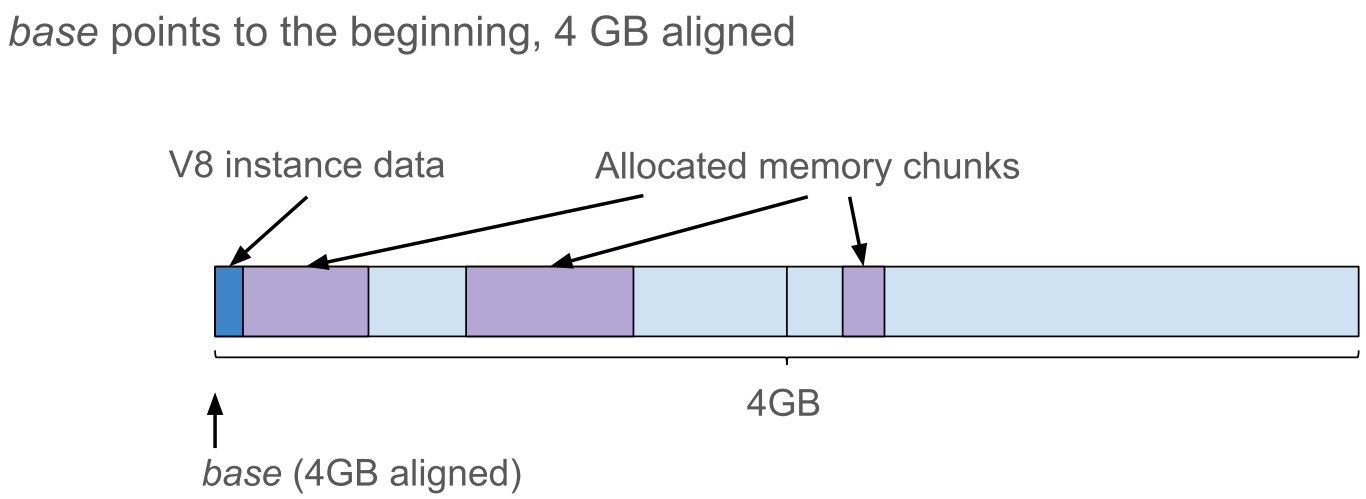


Trivial heap layout

Unfortunately, this is not an option for V8 since Chrome's renderer process may need to create multiple V8 instances in the same renderer process, for example for Web/Service Workers. Otherwise, with this scheme all these V8 instances compete for the same 4-GB address space and thus there is a 4-GB memory limit imposed on all V8 instances together.

Heap layout, v1

If we arrange V8's heap in a contiguous 4-GB region of address space somewhere else, then an **unsigned** 32-bit offset from the base uniquely identifies the pointer.



Heap layout, base aligned to start

If we also ensure that the base is 4-GB-aligned then the upper 32 bits are the same for all pointers:

```
            |----- 32 bits -----|----- 32 bits -----|
Pointer:    |________base_______|______offset_____w1|
```

We can also make Smis compressible by limiting the Smi payload to 31 bits and placing it to the lower 32 bits. Basically, making them similar to Smis on 32-bit architectures.

```
          |----- 32 bits -----|----- 32 bits -----|
Smi:      |ssssssssssssssssss|____int31_value___0|
```

where *s* is the sign value of the Smi payload. If we have a sign-extended representation, we are able to compress and decompress Smis with just a one-bit arithmetic shift of the 64-bit word.

Now, we can see that the upper half-word of both pointers and Smis is fully defined by the lower half-word. Then, we can store just the latter in memory, reducing the memory required for storing tagged value by half:

```
                    |----- 32 bits -----|----- 32 bits -----|
Compressed pointer:                     |______offset_____w1|
Compressed Smi:                         |____int31_value___0|
```

Given that the base is 4-GB-aligned, the compression is just a truncation:

```
uint64_t uncompressed_tagged;
uint32_t compressed_tagged = uint32_t(uncompressed_tagged);
```

The decompression code, however, is a bit more complicated. We need to distinguish between sign-extending the Smi and zero-extending the pointer, as well as whether or not to add in the base.

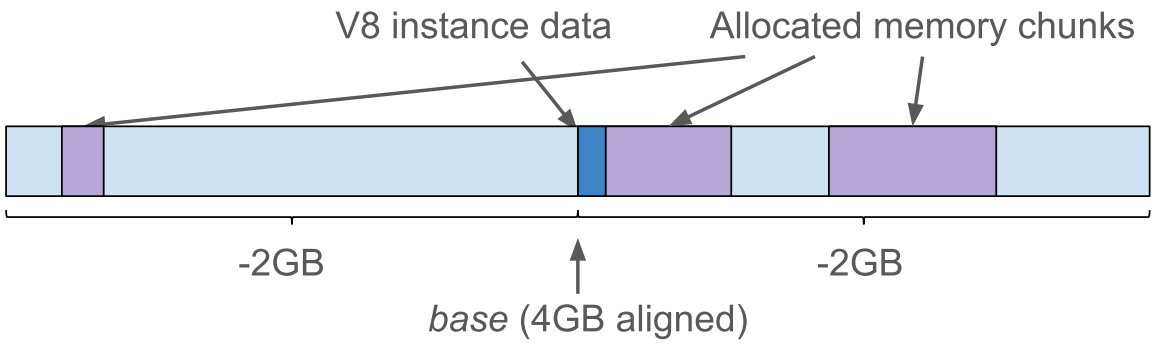
```
uint32_t compressed_tagged;

uint64_t uncompressed_tagged;
if (compressed_tagged & 1) {
  // pointer case
  uncompressed_tagged = base + uint64_t(compressed_tagged);
} else {
  // Smi case
  uncompressed_tagged = int64_t(compressed_tagged);
}
```

Let's try to change the compression scheme to simplify the decompression code.

Heap layout, v2

If instead of having the base at the beginning of the 4 GB we put the base in the *middle*, we can treat the compressed value as a **signed** 32-bit offset from the base. Note that the whole reservation is not 4-GB-aligned anymore but the base is.

base points to the middle, 4 GB aligned



Heap layout, base aligned to the middle

In this new layout, the compression code stays the same.

The decompression code, however, becomes nicer. Sign-extension is now common for both Smi and pointer cases and the only branch is on whether to add the base in the pointer case.

```
int32_t compressed_tagged;

// Common code for both pointer and Smi cases
int64_t uncompressed_tagged = int64_t(compressed_tagged);
if (uncompressed_tagged & 1) {
  // pointer case
  uncompressed_tagged += base;
}
```

The performance of branches in code depends on the branch prediction unit in the CPU. We thought that if we were to implement the decompression in a branchless way, we could get better performance. With a small amount of bit magic, we can write a branchless version of the code above:

```
int32_t compressed_tagged;

// Same code for both pointer and Smi cases
int64_t sign_extended_tagged = int64_t(compressed_tagged);
int64_t selector_mask = -(sign_extended_tagged & 1);
// Mask is 0 in case of Smi or all 1s in case of pointer
```

```
int64_t uncompressed_tagged =
    sign_extended_tagged + (base & selector_mask);
```

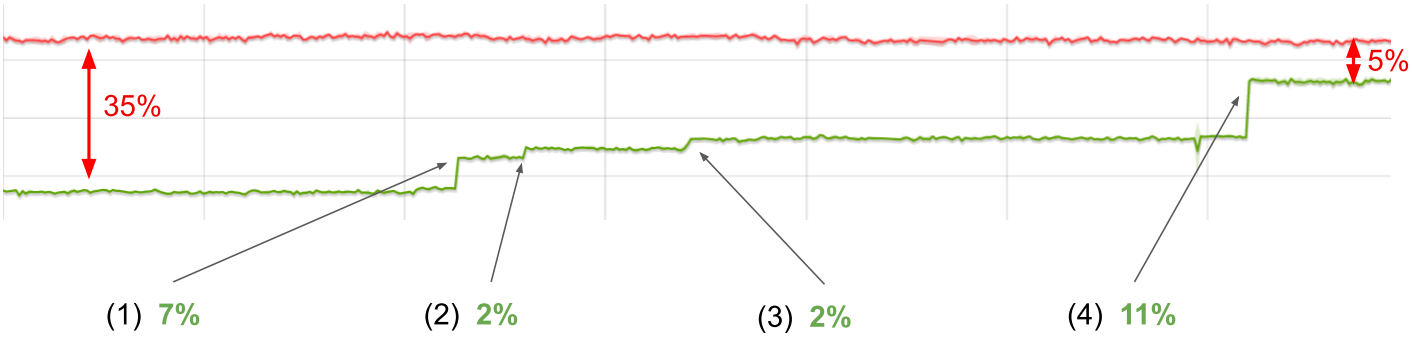
Then, we decided to start with the branchless implementation.

Performance evolution

Initial performance

We measured performance on Octane — a peak-performance benchmark we have used in the past. Although we are no longer focusing on improving peak performance in our day-to-day work, we also don't want to regress peak performance, particularly for something as performance-sensitive as *all pointers*. Octane continues to be a good benchmark for this task.

This graph shows Octane's score on x64 architecture while we were optimizing and polishing the Pointer Compression implementation. In the graph, higher is better. The red line is the existing full-sized-pointer x64 build, while the green line is the pointer compressed version.



First round of Octane's improvements

With the first working implementation, we had a ~35% regression gap.

Bump (1), +7%

First we validated our "branchless is faster" hypothesis, by comparing the branchless decompression with the branchful one. It turned out that our hypothesis was wrong, and the branchful version was 7% faster on x64. That was quite a significant difference!

Let's take a look at the x64 assembly.

Decompression	Branchless	Branchful
Code	`movsxlq r11,[…]` `movl r10,r11`	`movsxlq r11,[…]` `testb r11,0x1`

Decompression	Branchless	Branchful

	`andl r10,0x1`
	`negq r10`
	`andq r10,r13`
	`addq r11,r10`
		`jz done`
		`addq r11,r13`
		`done:`

Summary	20 bytes	13 bytes
	6 instructions executed	3 or 4 instructions executed
	no branches	1 branch
	1 additional register	

r13 here is a dedicated register used for the base value. Notice how the branchless code is both bigger, and requires more registers.

On Arm64, we observed the same - the branchful version was clearly faster on powerful CPUs (although the code size was the same for both cases).

Decompression	Branchless	Branchful

Code	`ldur w6, […]`
	`sbfx x16, x6, #0, #1`
	`and x16, x16, x26`
	`add x6, x16, w6, sxtw`
		`ldur w6, […]`
		`sxtw x6, w6`
		`tbz w6, #0, #done`
		`add x6, x26, x6`
		`done:`

Summary	16 bytes	16 bytes
	4 instructions executed	3 or 4 instructions executed
	no branches	1 branch
	1 additional register	

On low-end Arm64 devices we observed almost no performance difference in either direction.

Our takeaway is: branch predictors in modern CPUs are very good, and code size (particularly execution path length) affected performance more.

Bump (2), +2%

TurboFan is V8's optimizing compiler, built around a concept called "Sea of Nodes". In short, each operation is represented as a node in a graph (See a more detailed version in this blog post). These nodes have various dependencies, including both data-flow and control-flow.

There are two operations that are crucial for Pointer Compression: Loads and Stores, since they connect the V8 heap with the rest of the pipeline. If we were to decompress every time we loaded a compressed value from the heap, and compress it before we store it, then the pipeline could just keep working as it otherwise did in full-pointer mode. Thus we added new explicit value operations in the node graph - Decompress and Compress.

There are cases where the decompression is not actually necessary. For example, if a compressed value is loaded from somewhere only to be then stored to a new location.

In order to optimize unnecessary operations, we implemented a new "Decompression Elimination" phase in TurboFan. Its job is to eliminate decompressions directly followed by compressions. Since these nodes might not be directly next to each other it also tries to propagate decompressions through the graph, with the hope of encountering a compress down the line and eliminating them both. This gave us a 2% improvement of Octane' score.

Bump (3), +2%

While we were looking at the generated code, we noticed that the decompression of a value that had just been loaded produced code that was a bit too verbose:

```
movl rax, <mem>   // load
movlsxlq rax, rax // sign extend
```

Once we fixed that to sign extend the value loaded from memory directly:

```
movlsxlq rax, <mem>
```

so got yet another 2% improvement.

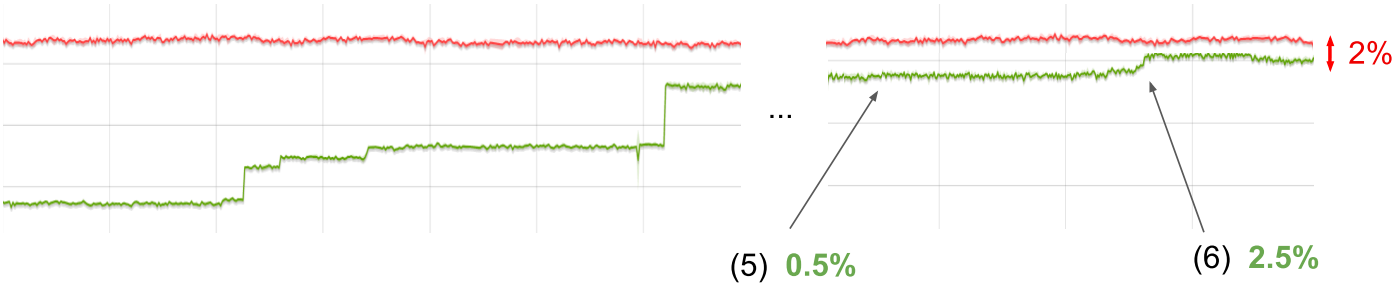
Bump (4), +11%

TurboFan optimization phases work by using pattern matching on the graph: once a sub-graph matches a certain pattern it is replaced with a semantically equivalent (but better) sub-graph or instruction.

Unsuccessful attempts to find a match are not an explicit failure. The presence of explicit Decompress/Compress operations in the graph caused previously successful pattern matching attempts to no longer succeed, resulting in optimizations silently failing.

One example of a “broken” optimization was allocation preternuring. Once we updated the pattern matching to be aware of the new compression/decompression nodes we got another 11% improvement.

Further improvements



Second round of Octane’s improvements

Bump (5), +0.5%

While implementing the Decompression Elimination in TurboFan we learned a lot. The explicit Decompression/Compression node approach had the following properties:

Pros:

- Explicitness of such operations allowed us to optimize unnecessary decompressions by doing canonical pattern matching of sub-graphs.

But, as we continued the implementation, we discovered cons:

- A combinatorial explosion of possible conversion operations because of new internal value representations became unmanageable. We could now have compressed pointer, compressed Smi, and compressed any (compressed values which we could be either pointer or Smi), in addition to the existing set of representations (tagged Smi, tagged pointer, tagged any, word8, word16, word32, word64, float32, float64, simd128).
- Some existing optimizations based on graph pattern-matching silently didn’t fire, which caused regressions here and there. Although we found and fixed some of them, the complexity of TurboFan continued to increase.

- The register allocator was increasingly unhappy about the amount of nodes in the graph, and quite often generated bad code.
- The larger node graphs slowed the TurboFan optimization phases, and increased memory consumption during compilation.

We decided to take a step back and think of a simpler way of supporting Pointer Compression in TurboFan. The new approach is to drop the Compressed Pointer / Smi / Any representations, and make all explicit Compression / Decompression nodes implicit within Stores and Loads with the assumption that we always decompress before loading and compress before storing.

We also added a new phase in TurboFan that would replace the “Decompression Elimination” one. This new phase would recognize when we don’t actually need to compress or decompress and update the Loads and Stores accordingly. Such an approach significantly reduced the complexity of Pointer Compression support in TurboFan and improved the quality of generated code.

The new implementation was as effective as the initial version and gave another 0.5% improvement.

Bump (6), +2.5%

We were getting close to performance parity, but the gap was still there. We had to come up with fresher ideas. One of them was: what if we ensure that any code that deals with Smi values never “looks” at the upper 32 bits?

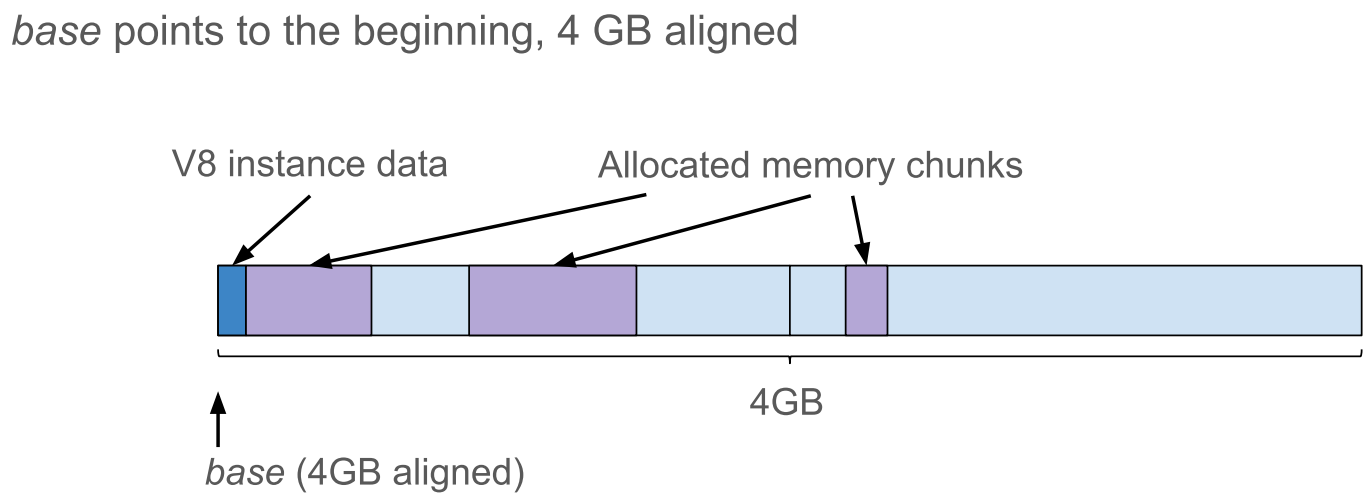
Let’s remember the decompression implementation:

```
// Old decompression implementation
int64_t uncompressed_tagged = int64_t(compressed_tagged);
if (uncompressed_tagged & 1) {
  // pointer case
  uncompressed_tagged += base;
}
```

If the upper 32 bits of a Smi are ignored, we can assume them to be undefined. Then, we can avoid the special casing between the pointer and Smi cases and unconditionally add the base when decompressing, even for Smis! We call this approach “Smi-corrupting”.

```
// New decompression implementation
int64_t uncompressed_tagged = base + int64_t(compressed_tagged);
```

Also, since we don't care about sign extending the Smi anymore, this change allows us to return to heap layout v1. This is the one with the base pointing to the beginning of the 4GB reservation.



Heap layout, base aligned to start

In terms of the decompression code, it changes a sign-extension operation to a zero-extension, which is just as cheap. However, this simplifies things on the runtime (C++) side. For example, the address space region reservation code (see the Some implementation details section).

Here's the assembly code for comparison:

Decompression	Branchful	Smi-corrupting

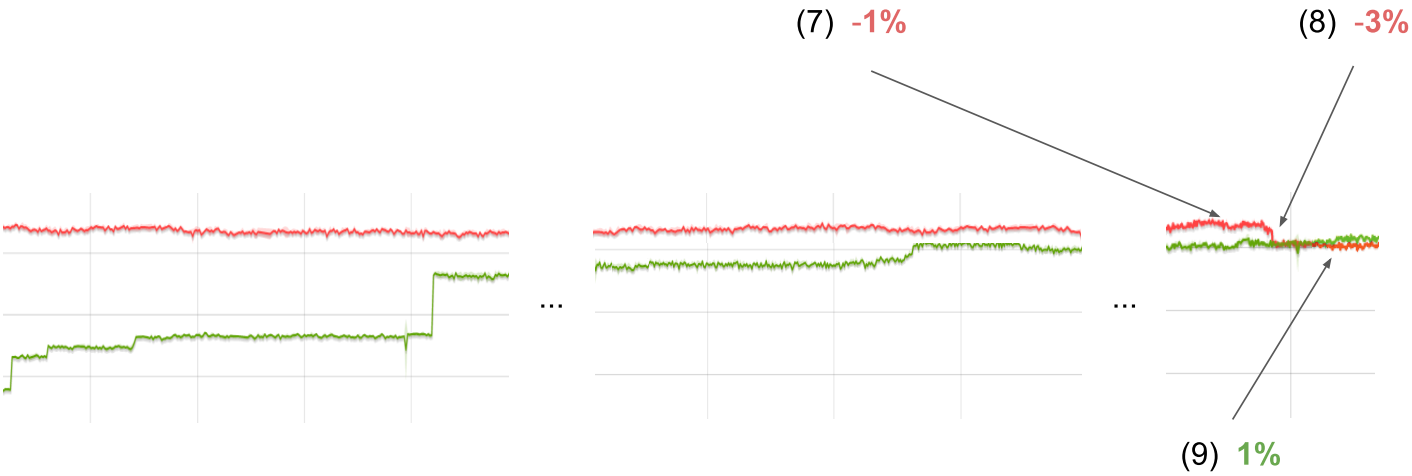
Code	`movsxlq r11,[…]`
	`testb r11,0x1`
	`jz done`
	`addq r11,r13`
	`done:`
		`movl r11,[rax+0x13]`
		`addq r11,r13`

Summary	13 bytes	7 bytes
	3 or 4 instructions executed	2 instructions executed
	1 branch	no branches

So, we adapted all the Smi-using code pieces in V8 to the new compression scheme, which gave us another 2.5% improvement.

Remaining gap

The remaining performance gap is explained by two optimizations for 64-bit builds that we had to disable due to fundamental incompatibility with Pointer Compression.



Final round of Octane's improvements

32-bit Smi optimization (7), -1%

Let's recall how Smis look like in full pointer mode on 64-bit architectures.

```
         |----- 32 bits -----|----- 32 bits -----|
Smi:     |____int32_value____|00000000000000000000|
```

32-bit Smi has the following benefits:

- it can represent a bigger range of integers without the need to box them into number objects; and
- such a shape provides direct access to the 32-bit value when reading/writing.

This optimization can't be done with Pointer Compression, because there's no space in the 32-bit compressed pointer due to having the bit which distinguishes pointers from Smis. If we disable 32-bit smis in the full-pointer 64-bit version we see a 1% regression of the Octane score.

Double field unboxing (8), -3%

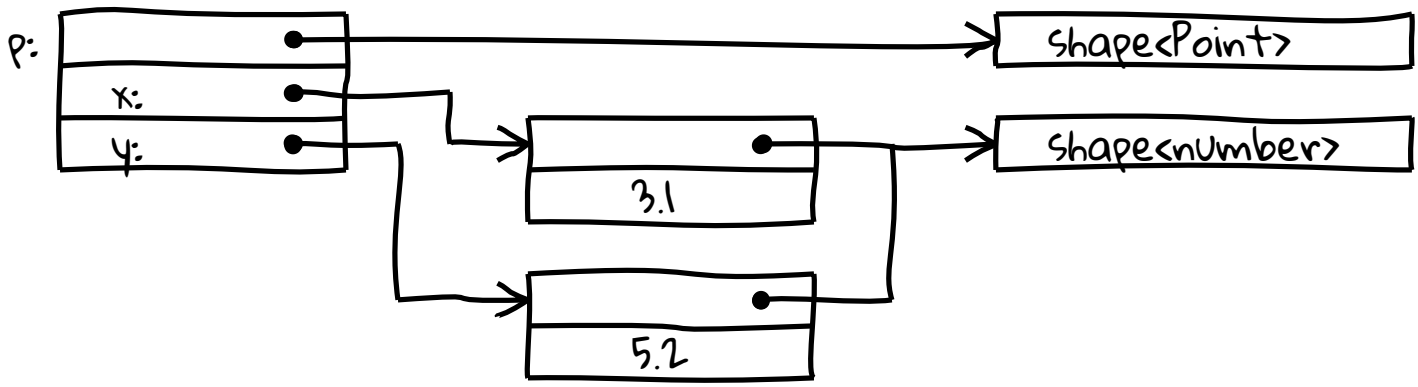
This optimization tries to store floating point values directly in the object's fields under certain assumptions. This has the objective of reducing the amount of number object allocations even more

than Smis do alone.

Imagine the following JavaScript code:

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
const p = new Point(3.1, 5.3);
```

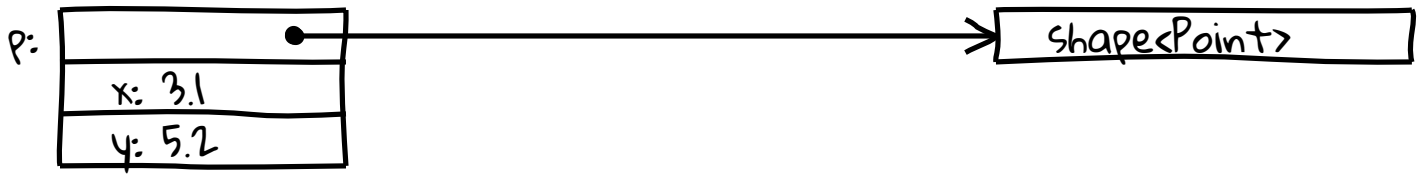
Generally speaking, if we look at how the object p looks like in memory, we'll see something like this:



Object p in memory

You can read more about hidden classes and properties and elements backing stores in this article.

On 64-bit architectures, double values are the same size as pointers. So, if we assume that Point's fields always contain number values, we can store them directly in the object fields.

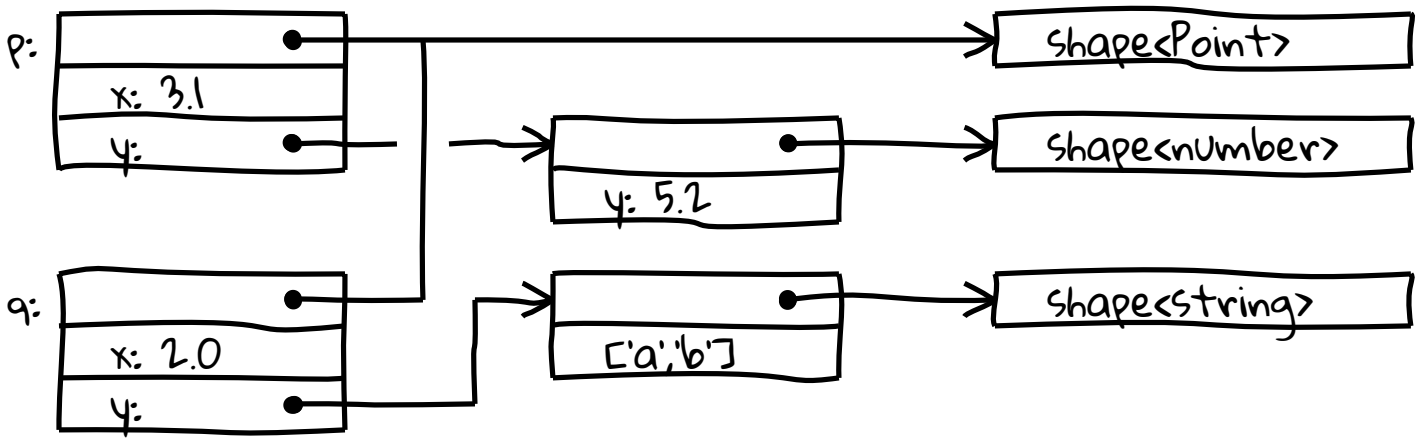


If the assumption breaks for some field, say after executing this line:

```
const q = new Point(2, 'ab');
```

then number values for the y property must be stored boxed instead. Additionally, if there is speculatively-optimized code somewhere that relies on this assumption it must no longer be used and must be thrown away (deoptimized). The reason for such a "field type" generalization is to minimize

the number of shapes of objects created from the same constructor function, which in turn is necessary for more stable performance.



Objects p and q in memory

If applied, double field unboxing gives the following benefits:

- provides direct access to the floating point data through the object pointer, avoiding the additional dereference via number object; and
- allows us to generate smaller and faster optimized code for tight loops doing a lot of double field accesses (for example in number-crunching applications)

With Pointer Compression enabled, the double values simply do not fit into the compressed fields anymore. However, in the future we may adapt this optimization for Pointer Compression.

Note that number-crunching code which requires high throughput could be rewritten in an optimizable way even without this double field unboxing optimization (in a way compatible with Pointer Compression), by storing data in Float64 TypedArrays, or even by using Wasm.

More improvements (9), 1%

Finally, a bit of fine-tuning of the decompression elimination optimization in TurboFan gave another 1% performance improvement.

Some implementation details

In order to simplify integration of Pointer Compression into existing code, we decided to decompress values on every load and compress them on every store. Thus changing only the storage format of tagged values while keeping the execution format unchanged.

Native code side

In order to be able to generate efficient code when decompression is required the base value must always be available. Luckily V8 already had a dedicated register always pointing to a “roots table” containing references to JavaScript and V8-internal objects which must be always available (for example, undefined, null, true, false and many more). This register is called “root register” and it is used for generating smaller and shareable builtins code.

So, we put the roots table into the V8 heap reservation area and thus the root register became usable for both purposes - as a root pointer and as a base value for decompression.

C++ side

V8 runtime accesses objects in V8 heap through C++ classes providing a convenient view on the data stored in the heap. Note that V8 objects are rather POD-like structures than C++ objects. The helper “view” classes contain just one uintptr_t field with a respective tagged value. Since the view classes are word-sized we can pass them around by value with zero overhead (many thanks to modern C++ compilers).

Here is an pseudo example of a helper class:

```
// Hidden class
class Map {
 public:
  …
  inline DescriptorArray instance_descriptors() const;
  …
  // The actual tagged pointer value stored in the Map view object.
  const uintptr_t ptr_;
};

DescriptorArray Map::instance_descriptors() const {
  uintptr_t field_address =
      FieldAddress(ptr_, kInstanceDescriptorsOffset);

  uintptr_t da = *reinterpret_cast<uintptr_t*>(field_address);
  return DescriptorArray(da);
}
```

In order to minimize the number of changes required for a first run of the pointer compressed version we integrated the computation of the base value required for decompression into getters.

```
inline uintptr_t GetBaseForPointerCompression(uintptr_t address) {
  // Round address down to 4 GB
  const uintptr_t kBaseAlignment = 1 << 32;
  return address & -kBaseAlignment;
}

DescriptorArray Map::instance_descriptors() const {
  uintptr_t field_address =
      FieldAddress(ptr_, kInstanceDescriptorsOffset);

  uint32_t compressed_da = *reinterpret_cast<uint32_t*>(field_address);

  uintptr_t base = GetBaseForPointerCompression(ptr_);
  uintptr_t da = base + compressed_da;
  return DescriptorArray(da);
}
```

Performance measurements confirmed that the computation of base in every load hurts performance. The reason is that C++ compilers don't know that the result of GetBaseForPointerCompression() call is the same for any address from the V8 heap and thus the compiler is not able to merge computations of base values. Given that the code consists of several instructions and a 64-bit constant this results in a significant code bloat.

In order to address this issue we reused V8 instance pointer as a base for decompression (remember the V8 instance data in the heap layout). This pointer is usually available in runtime functions, so we simplified the getters code by requiring an V8 instance pointer and it recovered the regressions:

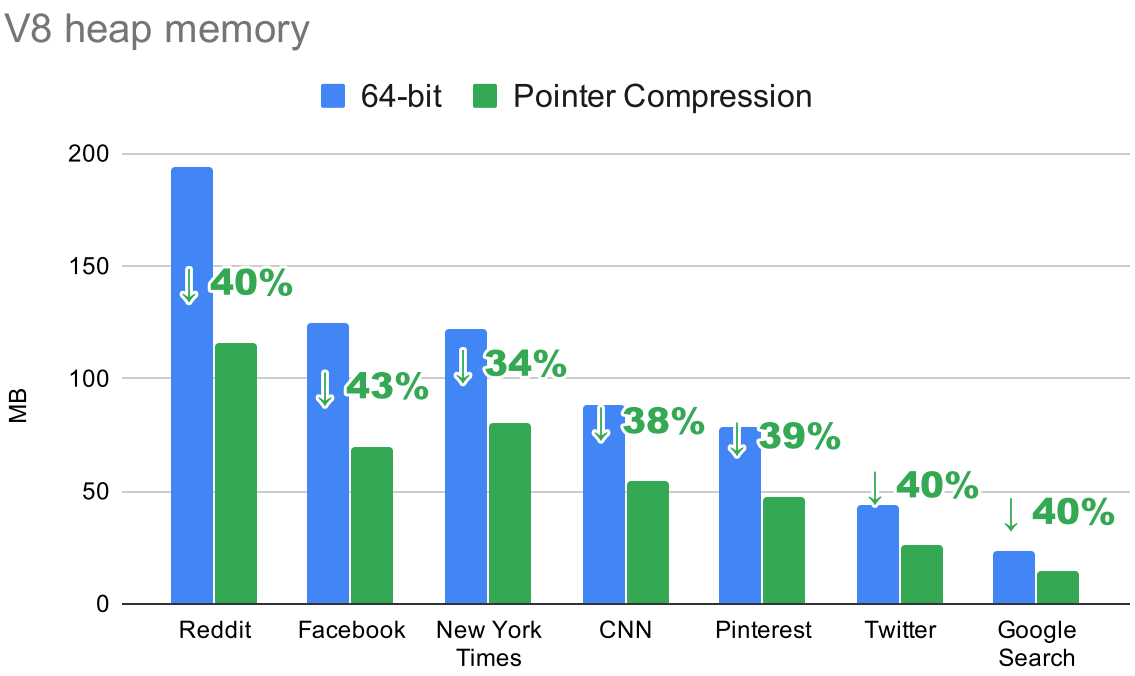
```
DescriptorArray Map::instance_descriptors(const Isolate* isolate) const {
  uintptr_t field_address =
      FieldAddress(ptr_, kInstanceDescriptorsOffset);

  uint32_t compressed_da = *reinterpret_cast<uint32_t*>(field_address);

  // No rounding is needed since the Isolate pointer is already the base.
  uintptr_t base = reinterpret_cast<uintptr_t>(isolate);
  uintptr_t da = DecompressTagged(base, compressed_value);
```

```
  return DescriptorArray(da);
}
```

Results

Let's take a look at Pointer Compression's final numbers! For these results, we use the same browsing tests that we introduced at the beginning of this blog post. As a reminder, they are browsing user stories that we found were representative of usage of real-world websites.

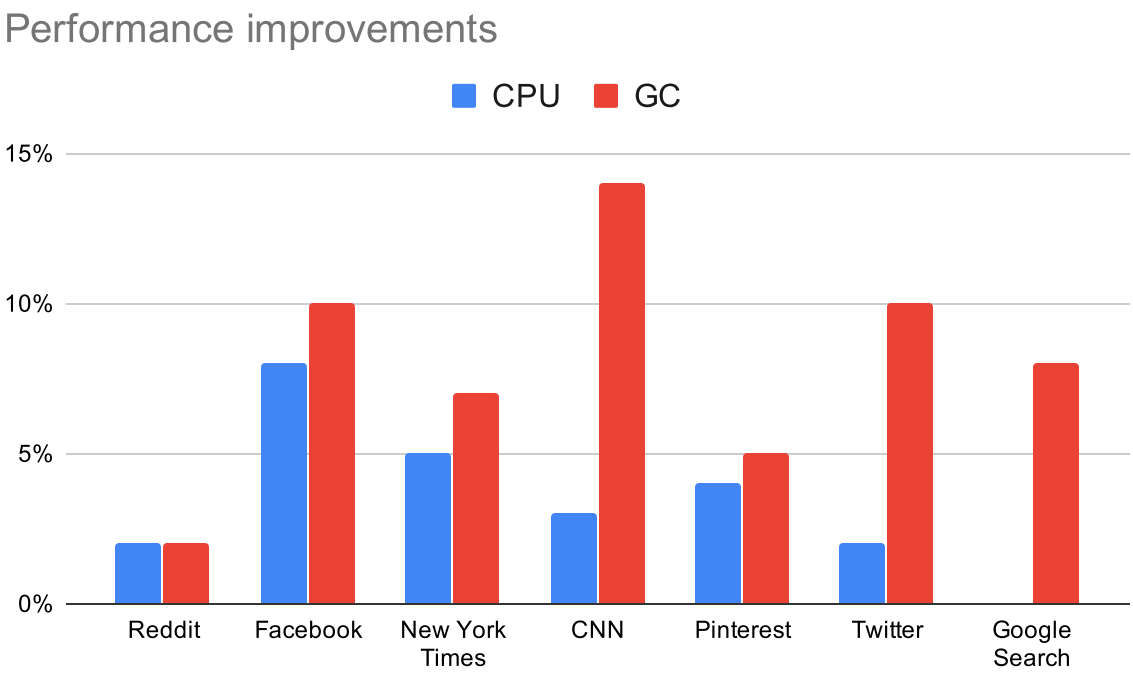
In them, we observed that Pointer Compression reduces **V8 heap size up to 43%**! In turn, it reduces **Chrome's renderer process memory up to 20%** on Desktop.



Memory savings when browsing in Windows 10

Another important thing to notice is that not every website improves the same amount. For example, V8 heap memory used to be bigger on Facebook than New York Times, but with Pointer Compression it is actually the reverse. This difference can be explained by the fact that some websites have more Tagged values than others.

In addition to these memory improvements we have also seen real-world performance improvements. On real websites we utilize less CPU and garbage collector time!



Improvements in CPU and garbage collection time

Conclusion

The journey to get here was no bed of roses but it was worth our while. 300+ commits later, V8 with Pointer Compression uses as much memory as if we were running a 32-bit application, while having the performance of a 64-bit one.

We are always looking forward to improving things, and have the following related tasks in our pipeline:

- Improve quality of generated assembly code. We know that in some cases we can generate less code which should improve performance.
- Address related performance regressions, including a mechanism which allows unboxing double fields again in a pointer-compression-friendly way.
- Explore the idea of supporting bigger heaps, in the 8 to 16 GB range.


Posted by Igor Sheludko and Santiago Aboy Solanes, *the* pointer compressors.

Retweet this article!