Turboshaft Late Load Elimination

Author: dmercadier@
Date: 10/07/2023
Last update: 25/07/2023
Visibility: public

This design document presents the new Late Load Elimination for Turboshaft, and in particular how the state is tracked: how we compute and track aliases, how we offer efficient lookup of the value at a memory location, how we efficiently invalidate whole memory regions/groups.

Background and Overview

The goal of Load Elimination is to remove loads when we statically know the result. For instance,

```
x[0] = 42;
return x[0];
```

can be optimized by the Load Elimination to

```
x[0] = 42;
return 42;
```

The main difficulty comes from the fact that any values could alias, and we thus have to always take a conservative approach. For instance, in this example:

```
x[0] = 42;
y[0] = 17;
return x[0];
```

we can't optimize the `return x[0]`, because the store to `y` could have overwritten the one to `x`.

Similarly, if we have:

```
x[0] = 0;
x[1] = 1;
x[2] = 2;
x[i] = 42
```

Upon reaching the last line (`x[i] = 42`), we have to "forget" everything we know about `x`, since this last store could have overwritten any of `x[0]`, `x[1]` or `x[2]` (or it could have written elsewhere).

We thus have 2 challenges to solve:

- Efficiently tracking the state, so that it's cheap to look up existing data, and it's also cheap to invalidate large amounts of data at once.
- Tracking aliases so that we have fewer things to invalidate

Efficient Lookup and Invalidation

We define a MemoryAddress as a tuple of `<OpIndex base, OpIndex index, int32 offset, uint8 element_size_log2, uint8 size>`, where the actual address is obtained by doing `base + offset + index * 2^element_size_log2` (and the stored/loaded value has size `size`).
Note that most stores and loads have an Invalid index and a valid offset. Loading/storing fields from/in an object for instance is always done with a constant offset rather than an index.
The main exception will be in loops, where it's common to have a loop iterator being used as an index (to load/store from/in an array).

Lookup

We maintain a global hashmap (called `all_keys_`), which maps from MemoryAddress to Snapshot Table Keys. From a Key, the Snapshot Table offers O(1) lookup of the value. We can thus set and lookup a value at a MemoryAddress in O(1) time.

Invalidation

Here are the following cases that we need to handle:

1. When writing at a base that doesn't alias with anything (see the Alias Tracking section) at a MemoryAddress with a valid Index, we need to invalidate everything at this base (because the value of the Index could be anything, and it could thus overwrite previous stores with invalid Index and valid Offsets.
2. When writing at an offset in a base that could alias, we need to invalidate
    a. Every store at the same offset, regardless of their base (except for bases that are known to have no aliases)
    b. Every stores at a valid Index (except for bases that are known to have no aliases)
3. When written at a valid Index in a base that could alias, we need to invalidate the whole state (except for bases that are known to have no aliases). We do the same thing when calling a function (since this function could a priori write anything anywhere in memory).

For 1, we need a way to efficiently access all of the Keys for a given base, and for 2, we need a way to efficiently access all of the Keys for a given offset.

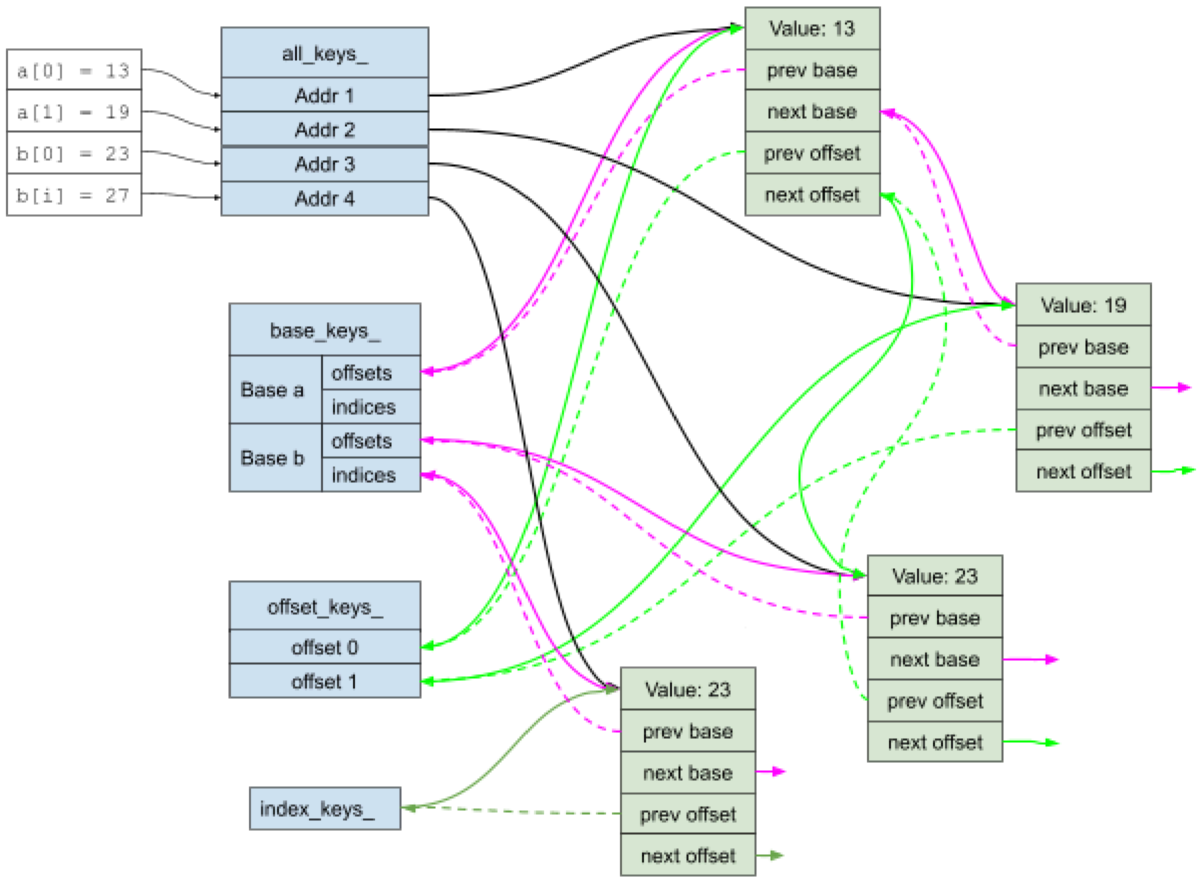
We thus have a Hashmap mapping each base to a linked list of all of the Keys for MemoryAddresses at this base (`base_keys_`), and another Hashmap mapping each offset to a doubly-linked list of all of the Keys for MemoryAddresses with this offset (`offset_keys_`). To reduce the memory overhead of these doubly-linked list, they are implemented intrusively in the

Keys themselves: each Key has prev_same_base, next_same_base, prev_same_offset, next_same_offset pointers. Furthermore, to be able to remove from these lists without having to do a lookup in base_keys_/offset_keys_ to lookup the list head, the prev pointers of each Key do not point to the previous Key, but rather to the corresponding next field of the previous Key (and to the head_ field of the List object in the case of the 1st item of the list). The implementation of these intrusive doubly linked lists was extracted to doubly-threaded-list.h.

MemoryAddresses with a valid Index are not stored in offset_keys_ though, since they need to be invalidated for stores at any offset. Instead, we keep them in a global doubly-linked list called index_keys_. This list uses the prev_same_offset and next_same_offset fields of the corresponding keys, since they are not otherwise used for MemoryAddresses with valid Indices.

When writing to a MemoryAddress with an invalid Index and a non-aliasing base, we just need to invalidate the exact same MemoryAddress and all of the MemoryAddresses at the same base that have valid Indices. To avoid having to iterate through all of the MemoryAddresses of this base in such cases, base_keys_ doesn't contain one doubly-linked list per base, but rather two: one called with_offsets, and the other one called with_indices. The former contains all of the MemoryAddresses with invalid Indices, and the latter contains all of the MemoryAddresses with valid Indices.

The following picture illustrates the state of these structures after a few stores have been performed (assuming that we knew that a and b didn't alias, and that 0 and i didn't alias either (otherwise we would only have b[i] = 27 in the state at the end)).
Dotted lines are used for "previous" pointers. Pointers related to base_keys_ are in purple, and pointers related to offset_keys_ are in green (and a darker green for the few pointers related to index_keys_).



Alias Tracking

Fresh Objects

Freshly allocated objects (ie, produced by an AllocateOp) never alias with anything. We thus keep track of these objects in a Snapshot Table called `non_aliasing_objects_`. We drop this information if:

- The object is stored somewhere (= it's the `value` of a StoreOp).
- The object is merged by a Phi
- The object is passed to a function call that has the `can_write` Effect.

Map Checks

Two objects with different maps can never alias. Thus, we keep track of object maps in the `object_maps_` Snapshot Table, and before invalidating for an offset/index store, we check the maps of the destination and of the potential base to invalidate: if they both have distinct known maps, then we do not invalidate.

Note that LateLoadElimination runs after MachineLoweringReducer, which removes the CheckMaps. To still preserve map information, we added a new AssumeMap operator, which MachineLoweringReducer inserts after the lowerings of CheckMaps.

Furthermore, storing all of the potential maps for an object would 1) require dynamically allocating zone memory and 2) could be a bit expensive when comparing the maps of 2 objects. We thus opted for a compact version of this: for each object, we just store the bitwise `or` and the bitwise `and` of all of their possible maps. If the `and` of object A has a bit set that the `or` of object B doesn't have set, then it means that all of the maps of A have a bit that none of the maps of B have, which means that objects A and B have different maps. (this logic implemented in the `MapMaskAndOr` class)

Performance

Turbofan's Load Elimination has a few limits to prevent excessive compile times on corner cases (such as the benchmark I'm using in this section):

- A limit on the number of fields tracked per object (32)
- A limit on the number of objects tracked (100)
- A limit on the number of total fields tracked (300)

The second and third limits are easy to change. However, the 1st one is used to have O(1) access to the fields of an object by offset: there is an array of values per object, where the value at offset *n* of the object is stored at offset *n* in this array. Increasing this limit increases memory usage (since there is an array of size 32 reserved for each base), which could also regress performance.
For this test, I've removed the 2nd and 3rd limit, and set the 1st one to 128 instead of 32 (which still means that a lot of fields won't be tracked, since objects will have several hundreds of fields).

A good function to compile to test Load Elimination is one that copies an object field-by-field, as the generated code will have to resize the backing store of the destination every 3 fields:

```
let out = {};
out.a0 = in.a0;
out.a1 = in.a1;
out.a2 = in.a2;
…
```

(see Issue 14133: Slow TF compilation for functions that do a lot of transition stores to the same object for details about this behavior)
I've compared the time needed for Turbofan's LoadElimination and this new Turboshaft LateLoadElimination, depending on the number of fields that we copy in the function (when testing Turboshaft's LateLoadElimination, I disabled Turbofan's LoadElimination, to make sure that Turboshaft doesn't benefit from it):

Fields copied	Turbofan time	Turboshaft time	Turboshaft speedup
100	5ms	0.5ms	x10
300	360ms	5ms	x72
400	1350ms	10ms	x135
500	3150ms	20ms	x156
600	6500ms	40ms	x162
700	10250ms	65ms	x157
800	16850ms	105ms	x160
900	25400ms	140ms	x181
1000	33500ms (1.56GB)	175ms (92MB)	x191

(note that the graph size is growing quadratically in the number of “fields copied”)

Keep in mind that Turbofan’s LoadElimination is still not eliminating most of the loads because of its limit on the number of fields per object tracked. As a result, the graph is larger that it could be, leading in worst compile time late in the pipeline, in particular during the AssignSpillSlots phase of the register allocator (1.3sec for 700 fields, 2.1sec for 800 fields, 2.9sec for 900 fields, 4.2sec for 1000 fields). With Turboshaft’s LoadElimination, this AssignSpillSlots phase never takes more than 15ms (although the AllocateGeneralRegisters phase becomes slightly more expensive (400ms instead of 200ms, since more things are in register rather than memory)).

I also benchmarked the generated code: with 700 fields, the code generated by Turbofan is about 1.66 times slower than the one generated by Turboshaft. (once again, this is because Turbofan does not actually track the whole state) (this does not include compilation time)

A note on the increased “number of fields per object tracked” limit of Turbofan: when setting it to its regular 32 (instead of the 128 I used for this benchmark), Load Elimination time for TS goes down to 10.5sec (instead of 33.5) and memory usage to 445MB (instead of 1.5GB). However, this means that even fewer loads are eliminated.

Small disclaimer still: Turbofan’s LoadElimination does a bit more than Turboshaft, since it also performs Check elimination (checks have been lowered by the time we reach Turboshaft’s LateLoadElimination, so it would make no sense to do it there as well), but this really is not the reason for the speed difference :D

Miscellaneous

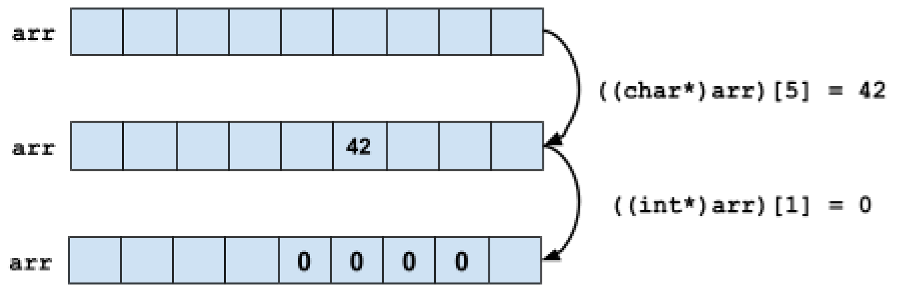
Overlapping Stores

TL;DR: we don't support TypedArrays/DataViews/ArrayBuffers, and other than that, stores can't overlap in any dangerous way.

Imagine that a graph does the equivalent of:

```
((char*)arr)[5] = 42;
((int*)arr)[1] = 0;
```

Which visually means:



The MemoryAddress for the 1st Store will be `{base:arr, offset:5, size:1}`, and for the 2nd it will be `{base:arr, offset:4, size:4}` (ignoring `Index` and `element_size_log2` which are not used here). Given the addresses, it looks like these Stores don't alias (since they have different offsets), which means that we won't invalidate the 1st one when processing the 2nd one, and a future Load from `((char*)arr)[5]` would incorrectly return 42.

There are 2 cases were this could happen in JavaScript:

- TypedArrays/DataViews
- Strings

TypedArrays / DataViews

There are two cases where TypedArrays/DataViews stores could overlap:

```
const u32s = Uint32Array.of(3, 8);
const u8s = new Uint8Array(u32s.buffer);
u32s[0] = 0xffffffff;
u8s[1] = 0; // Overlaps with previous store!
```

And:

```
let buffer = new ArrayBuffer(10000);
let ta1 = new Int32Array(buffer, 0/*offset*/);
let ta2 = new Int32Array(buffer, 100*4/*offset*/);
ta2[0] = 0xff;
ta1[100] = 42; // Same destination as the previous store!
```

The 1st case could be dealt with: invaliding an address during LoadElimination could invalidate every other (potentially aliasing) addresses at the same offset as well as with offsets from -7 to +7 (the maximum size of array elements is 8, so things can't alias in this way with elements that are further than 7 offsets away).
However, the 2nd case can't really be dealt with: any 2 stores with different bases could alias regardless of their offsets. We have thus no choice but to not perform Load Elimination for TypedArrays/DataViews.

In the Turboshaft graph, all TypedArray Stores and Loads will be StoreTypedElement and LoadTypedElement (LoadDataViewElement/StoreDataViewElement for DataViews). They are lowered to regular Stores/Loads during MachineLoweringReducer, which runs before LateLoadElimination, and I added a `always_canonically_accessed` field to `LoadOp::Kind`, which we set to true for TypedArray/DataView loads/stores during this lowering (for regular JS stores/loads, this will be false).
During LateLoadElimination, when we encounter a Store offset `x` with a `can_overlap` bit set to true, we ignore it.

Strings

The only other case of overlapping Stores is for Strings: Strings are 0-padded on the right, so that we can efficiently compare them 32-bit by 32-bit. What could happen is a 32-bit Store of 0, followed by other stores of single characters (8 or 16 bits), followed by a read (32-bit if for a comparison, or 8/16-bit if for something else). This is not an issue with the current implementation, since we will never try to load from the string before we've finished writing it, and it's guaranteed that at least one of the character Stores will invalidate the padding Store (because at least the 1st byte of the 32-bit padding has to be a character of the string, otherwise we would not have allocated these 32 bits).