TurboFan Developer Tools Integration

Attention: Shared Google-externally

Author: bmeurer@chromium.org, tebbi@chromium.org, yangguo@chromium.org

Last updated: 2017/02/21

This document describes the various design and implementation bits relevant for proper integration of the TurboFan compiler pipeline (including the tiny details on the boundary to the other subsystems, i.e. the Ignition interpreter) with the various developer tools supported by V8 (i.e. Chrome DevTools, Turbolizer, IrHydra 2, or about:tracing).

Accurate source positions for --trace-deopt

When TurboFan hits a soft or eager deoptimization exit, it instructs the `Deoptimizer` to tear down the current optimized activation record and replace it with appropriate activation records for the fullcodegen or Ignition tiers. Each optimized activation can include any number of inlined activations. These are described in the `DeoptimizationData` that is attached to each optimized `Code` object, and the `Deoptimizer` knows how to extract the relevant data.

When deoptimizing from optimized code back to unoptimized code, TurboFan (and Crankshaft) won't necessarily restore the state at exactly the corresponding position in unoptimized code. Instead TurboFan can decide to take an earlier `Checkpoint` (or `FrameState`) in unoptimized code, as long as there was no observable side effect since then. For example consider the following function `foo`:

```
function foo(a, b) {
  var x = a + 1;
  return x * b;
}
```

And assume that we have `Signed32` feedback for `a + 1` and `x * b`. So we will lower both to integer operations, assuming that the inputs (in these case `a` and `b`) are in 32-bit integer range, and the results of the addition and the multiplication don't overflow the 32-bit integer range. But since neither the addition nor the multiplication can have observable side effects then (i.e. changing some global variables or storing a file to disk), TurboFan will only have a single `Checkpoint` in this function, right at the beginning, and all failing checks that guard optimistic assumptions will resume execution in unoptimized code at the very beginning of `foo`.

Looking at the initial schedule of `foo` you can see that there's only a single `Checkpoint` right at the beginning of the function:

```
--- BLOCK B0 ---
  0: Start
  3: Parameter[1](0)
  2: Parameter[%this#0](0)
  4: Parameter[2](0)
  1: Parameter[%context#5](0)
  9: Parameter[%closure#-1](0)
  5: HeapConstant[0x3d9ea8282311 <undefined>]
  31: HeapConstant[0x3d9ea8284b41 <Odd Oddball: optimized_out>]
  8: TypedStateValues[kRepTagged|kTypeAny](5)
  32: TypedStateValues[kRepTagged|kTypeAny](31)
  6: TypedStateValues[kRepTagged|kTypeAny, kRepTagged|kTypeAny, kRepTagged|kTypeAny](2, 3,
4)
  10: FrameState[INTERPRETED_FRAME, 0, Ignore, 0x2971560abbf9 <SharedFunctionInfo foo>](6, 32,
8, 1, 9, 0)
  12: Checkpoint(10, 0, 0)
  45: Int64Constant[0]
  44: ExternalConstant[0x7f13eb1118e8]
  47: LoadStackPointer
  46: Load[kRepWord64](44, 45, 12, 0)
  48: Uint64LessThan(46, 47)
  49: Branch[True](48, 0) -> B2, B1
--- BLOCK B1 (deferred) <- B0 ---
  51: IfFalse(49)
  15: FrameState[INTERPRETED_FRAME, 1, Ignore, 0x2971560abbf9 <SharedFunctionInfo foo>](6,
32, 8, 1, 9, 0)
  55: Int32Constant[0]
  54: ExternalConstant[0x7f13e9238580<StackGuard.entry>]
  56: HeapConstant[0x3d12b74042e1 <Code: STUB>]
  13: Call[Code:StackGuard:r1s0i4f1t0](56, 54, 55, 1, 15, 12, 51)
  14: IfSuccess(13)
  Goto -> B3
--- BLOCK B2 <- B0 ---
  50: IfTrue(49)
  Goto -> B3
--- BLOCK B3 <- B2, B1 ---
  52: Merge(50, 14)
  53: EffectPhi(12, 13, 52)
  42: CheckedTaggedSignedToInt32(3, 53, 52)
  43: Int32Constant[1]
  19: CheckedInt32Add(42, 43, 42, 52)
  41: CheckedTaggedSignedToInt32(4, 19, 52)
  26: CheckedInt32Mul[check-for-minus-zero](19, 41, 41, 52)
  40: ChangeInt32ToTagged(26)
  29: Return(40, 26, 52) -> B4
--- BLOCK B4 <- B3 ---
  30: End(29)
```

So whenever we deoptimize from one of the checks that come later, i.e. all the `Checked` operators scheduled in `B3`, we resume at the `FrameState #10` with with the bytecode offset 0. Now looking at the bytecode that means we resume execution in Ignition right before the `StackCheck`:

```
[generating bytecode for function: foo]
Parameter count 3
Frame size 8
 3154 E> 0x2971560ac4a5 @    0 : 73                StackCheck
 3175 S> 0x2971560ac4a6 @    1 : 33 01 03 02       AddSmi [1], a0, [2]
         0x2971560ac4aa @    5 : 1e fa             Star r0
 3182 S> 0x2971560ac4ac @    7 : 1d 02             Ldar a1
 3193 E> 0x2971560ac4ae @    9 : 2a fa 03          Mul r0, [3]
 3196 S> 0x2971560ac4b1 @   12 : 76                Return
```

So when something weird happens, i.e. `a` or `b` are not `Smi`s, but some object that actually has an `@@toPrimitive` method installed, that would throw, we'd go back to the interpreter and execute everything appropriately and also generate the exception from the correct source position in the Interpreter. However when we deoptimize to the interpreter, we ideally want to know exactly what caused the deoptimization, for example for `--trace-deopt` output or for use by DevTools or IrHydra 2. In that case it is not sufficient to just take the source position from the bytecode that corresponds to the byte code offset that we deoptimize to, as that would mean that deoptimizations caused by `x * b` would also point to the beginning of the function, that is to the line `var x = a + 1`.

So we need to record additional, accurate source positions for each check that guards an optimistic assumption in optimized code. This is done by adding a special `DEOPT_POSITION` entry to the `RelocInfo` for the `Deoptimizer` entry points (i.e. the guard code generated for the checks), in addition to the `DEOPT_REASON` which gives a hint about why the check is failing. This deoptimization source position is threaded through the various stages of the compilation pipeline and finally embedded into the `Code` object during code generation.

As long as there's no inlining involved, this source position easily identifies the exact position inside the script of the function that was optimized. However with inlining we need a second dimension that identifies the function (or script) relative to which we need to interpret the character offset.

Ideally, we do not only want the original position of a deopt reason in an inlined function, but also information where it was inlined, that is, the virtual call-stack of inlinings leading to this position. The `Deoptimizer` contains enough information to reconstruct the call stack of a `Checkpoint`, but this is insufficient. Turbofan can freely move deoptimization checks to earlier positions, even before side-effects or out of inlined function calls. So it can happen that the position of a deopt reason is at a different inlining level than the `Checkpoint` the

`Deoptimizer` reconstructs the call-stack of. Thus we need additional data to associate source positions with precise inlining stacks.

Source positions for the bytecode pipeline

Owner: neis@chromium.org
Status: Mostly done, looking for ways to properly test this.
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5439

At the time of this writing only the `AstGraphBuilder` supports attaching source position information to `Node`s via the `AstGraphBuilderWithPositions` shim in `pipeline.cc`. We will need to add something similar for the `BytecodeGraphBuilder`. It shouldn't be too hard to add (we have the source-position-table on the bytecode array), but we will need to add it before any of this works for Ignition.

Unfortunately, the only tests for these source positions are in `cctest/test-cpu-profiler.cc`. These tests are flaky, and throughout cctest.status you can see they are disabled based on platform, architecture, time of day, etc. It's probably a good high level code cleanup task to make these work everywhere. New tests should be part of the fix.

Source positions for inlined functions in optimized code

Status: **Done**
Owner: tebbi@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5432

At the moment, there are two different classes for source positions with different encodings: `internal::SourcePosition` and `compiler::SourcePosition`. We merge `compiler::SourcePosition` and `internal::SourcePosition` to a single class used throughout the codebase. The new `internal::SourcePosition` instances store an id identifying an inlined function in addition to a script offset. `SourcePosition::InliningId()` refers to a the new table DeoptimizationInputData::InliningPositions(), which provides the following data for every inlining id:

- The inlined SharedFunctionInfo as an offset into DeoptimizationInfo::LiteralArray
- The SourcePosition of the inlining. Recursively, this yields the full inlining stack.

Before the Code object is created, the same information can be found in CompilationInfo::inlined_functions().

If SourcePosition::InliningId() is SourcePosition::kUnknown (-1), it refers to the outer (non-inlined) function.
So every SourcePosition has full information about its inlining stack, as long as the corresponding Code object is known. The internal representation of a source position is a positive 64bit integer.

All compilers create now appropriate source positions for inlined functions. In the case of Turbofan, this required using AstGraphBuilderWithPositions for inlined functions too. So this class is now moved to a header file.

At the moment, the additional information in source positions is only used in --trace-deopt and --code-comments. The profiler needs to be updated, at the moment it gets the correct script offsets from the deopt info, but the wrong script id from the reconstructed deopt stack, which can lead to wrong outputs. This should be resolved by making the profiler use the new inlining information for deopts.

By changing the inlined deoptimization tests in test-cpu-profiler.cc, we can enable them for Turbofan, changing them to a case where the deopt stack and the inlining position agree. It is currently still broken for other cases.

The following additional changes are necessary:

- The source position table (internal::SourcePositionTableBuilder etc.) supports now 64bit source positions.
- The class HPositionInfo was effectively dead code and is now removed.
- SourcePosition has new printing and information facilities, including computing a full inlining stack.
- We had to rename compiler/source-position.{h,cc} to compiler/compiler-source-position-table.{h,cc} to avoid clashes with the new src/source-position.cc file.
- The new wrapper PodArray for ByteArray is a template working with any POD-type. This is used in DeoptimizationInputData::InliningPositions().
- We remove HInlinedFunctionInfo and HGraph::inlined_function_infos, because they were only used for the now obsolete Crankshaft inlining ids.
- Crankshaft managed a list of inlined functions in Lithium: LChunk::inlined_functions. This is an analog structure to CompilationInfo::inlined_functions. To register the offsets into the literal array in a uniform way, we remove LChunk::inlined_functions and make Crankshaft use CompilationInfo::inlined_functions instead. This is a safe change because LChunk::inlined_functions has no other uses and the functions in CompilationInfo::inlined_functions have a strictly longer lifespan, being created earlier (in Hydrogen already).

Accurate source positions for the sampling profiler

The sampling profiler couples all profiling information to its own mirror of Code objects. So support for inlined source positions requires breaking this coupling because it means that one code object can contribute to the counts for another function/code object. It is unclear where to count inlined ticks: the inlining code object (what happens right now) or another optimized code object of the inlined function (which might not exist and seems also wrong).

Accurate source positions for allocation profiling

In order for the heap allocation profiler to correctly identify the function that allocated a certain object we'd need to have at least some kind of mapping from the program counter of the deferred allocation runtime call back to the function that originally contained the expression that was lowered to the allocation. This is only relevant for the inlining case.

However there might be a need to even extract accurate source position for a certain allocation site in optimized, which would require also mapping allocation sites to source positions (including inlining information).

Allocation folding vs. heap allocation profiling

Optimizations like escape analysis and allocation folding might break heap allocation profiling or at least make the data unreliable. Especially allocation folding doesn't play well with the current approach to treat calls to the runtime `%AllocateInNewSpace` and `%AllocateInTargetSpace` methods as returning a single new object. This is not the case with allocation folding (i.e. as implemented in the `MemoryOptimizer` phase), which reserves one big chunk of memory and then allocates into this chunk without letting the runtime track individual allocations. This reservation in the case that bump pointer allocation would fail, i.e. when we need to go to the runtime for allocation, works by requesting space for the upper bound of the folded allocations from the runtime (in either new or old space) and immediately after returning from the runtime function, resetting the `top` pointer to the assumed object start and bump pointer allocating from there (without having to check against the limit again for everything in that particular `AllocationGroup`).

That means the runtime has no way to reliably track individual object allocations or precise allocated sizes at the point of allocation. Instead the runtime can only determine the amount of memory allocated since the last time it had to update the allocation top pointer, but there's no way to reliably associate that with functions that do allocate or accurate source position information within those functions.

General accurate source positions information

To be clarified: Is there any use-case for source position information beyond the concrete use cases outlined above?

Practical Policy for Ongoing Implementation (formulated with yangguo@chromium.org)

If profiling is on, we should have access to the richest possible information. For our own investigations, we really want to have good deopt source position information in optimized code. This is expensive, so we don't want it on by default.

If you want to profile late, then you'll have to deopt all optimized code or restart the app or web page to turn on profiling information.

Work items (neis@chromium.org is driving this):

- Let's make all turbofan decisions regarding source information depending on whether SharedFunctionInfo::kSourcePositionsEnabled.
- --trace-deopt - this should enable kSourcePositionsEnabled
- --trace-turbo - this should enable kSourcePositionsEnabled
- Enable this in pipeline.cc when profiling is on (this is in the isolate). Also when heap profiling is on.
- Get rid of --turbo-source-positions