Serializer for background compilation

Authors: mslekova@chromium.org, jarin@chromium.org, neis@chromium.org

Last Updated: 2018/11/19

Please note: shared publicly

Tracking bug: https://crbug.com/v8/7790

Motivation

This is a follow-up effort coming after experimenting with and reasoning about the effects of implementing a Bytecode copying phase. The naive implementation resulted in average of 5.5% regression on Octane 2, which led to a number of optimization experiments. Unfortunately none of them led to significant recovery, which means we'll need to implement a slightly different approach. Our proposal is to introduce a new, lightweight phase that will do bytecode walking and copy data that will be later needed for compilation.

Overview

The general idea is to introduce a bytecode walker as part of the Serialization phase, that will use heuristics to decide which data will later be needed for compilation and copy this data. The class will build some lightweight alternative of the BytecodeGraphBuilder::Environment and will "learn" some insights from the bytecode. For instance for Load* bytecodes it will learn both from the receiver and the result of the load operator.

High-level design

We'll add a new class named **SerializerForBackgroundCompilation**. It will have a simple public API with just one method - **Run**. It will set up an **Environment** and a **BytecodeIterator**, then kick in **TraverseBytecode** method, which uses the iterator and delegates the processing of a single bytecode to bytecode-specific methods (e.g. VisitLdaConstant, VisitCall, etc.). The TraverseBytecode method will be recursive, triggering copying of data for possible inline candidates.

The *environment* will be a zone-allocated object which keeps track of the current state (of the parameters and local variables) necessary for the heuristic decisions. For each parameter and register, it will provide additional feedback. This might be a set of maps seen until now, a set of constant values or a set of FunctionBlueprints. It will also keep a pointer to its parent environment, if the function was called from another optimized function (essentially Call* visitors

will build the hierarchy). The FunctionBlueprint is a structure holding a pair of SharedFunctionInfo and FeedbackVector for the cases where the JSFunction object is not created yet.

Here's a tentative declaration of the environment class and pseudo-code usage example for it, implementing the TraverseBytecode method. More details need to be added to the merging/recursive logic, specifically for handling calls and loops.

The *traversal* will generally be linear, updating the Environment accordingly. For instance, the Lda* visitor will set the environment's accumulator value to point to the function loaded, VisitStar will copy the environment record for the accumulator into the record for the register, and VisitCall will then read the record for the target register, and will see that we are calling particular function.
Eventually the SerializerForBackgroundCompilation class will handle all copies of data needed for inlining and the other compiler phases, such as copies of BytecodeArray, FeedbackVector, SharedFunctionInfo, ConstantPoolEntry's and the JSFunction's themselves.

An *enhancement* of the heuristics detecting for future inlining candidates should be based on the following stages, each of them leading to the function in question being copied and traversal continuing recursively for it:

1. A constant is encountered and this constant has a JSFunction value;
2. A function is encountered with the same SharedFunctionInfo and FeedbackVector as before, even though the function itself is not a constant.

Testing

A few levels of testing could be implemented to ensure easy maintenance of the Serializer:

Unit tests

- Write cctests, taking JS as input, running the Serializer on it and checking that a given function `IsSerializedForCompilation()`; *(the easiest to start with)*
- Write cctests, transforming given bytecode sequences to expected hints; *(see Considerations below)*
- Add cctest(s) for correct branching and merging of Environments; *(the implementation is not there yet)*

Integration tests

- Write mjsunit tests from simplified examples for missed inlining opportunities (e.g. taken from RayTrace on Octane, which is the one with the heaviest performance penalty)

Considerations

- We could use the CompileRun(source) helper in cctest, but this would limit us to a single function, while we ultimately want to test w.r.t. Inlining, which means interaction between functions.
- We could use the FunctionTester as in test-js-context-specialization, but this will trigger the whole compilation machinery, so it won't be a unit test anymore.
- We could use the private constructor of SerializerForBackgroundCompilation, which expects FunctionBlueprint instead of JSFunction. Then we could add other helpers (e.g. CreateSFI(bytecode string) and pair it with an empty FeedbackVector. This approach might imply creating additional dummy objects, members of the BytecodeArray, e.g. constant_pool, handler_table, source_position_table.

Alternatives considered

We did some experimentation w.r.t the internal structure used for storing the hints in the environment, which turned out to be a bottleneck once cloning/merging of Environments was implemented to support Jump bytecodes. In particular, ZoneVector, ZoneSet and ZoneUnorderedSet were considered. The following table compares the performance and memory footprint of the different implementations when the `test/mjsunit/es6/spread-call.js` is executed with using `--turbo-stats` flag, where baseline is the implementation that doesn't handle Jumps (before this CL).

	Turbofan phase	Time (% of total)	Space in bytes (%)
baseline	serialize bytecode	1.0%	552760 (2.4%)
Vector	serialize bytecode	1.1%	716824 (3.1%)
Set	serialize bytecode	1.2%	781752 (3.4%)
Unordered set	serialize bytecode	12.2%	15207592 (40.8%)

Work log

- Implemented simple inlining success ratio (# of missed opportunities due to JSFunctionRef not serialized / total # of attempts). Results can be found here.
- PS4 - Added basic support for single Lda/Sta/Call sequence;
- PS7 - Added an "extra_serialized" flag and filtered inlining based on this one. The regression can be seen in the "Filter extra" column here;

- PS9 - Supported one more bytecode sequence for raytrace#closure - RayTrace recovered by 8%;
- PS10 - Add support for almost all Lda/Call bytecodes and a few Sta - total score recovery of 2%;
- PS11 - Add support for storing properties (correctness fix);
- PS14 - Enhance Call handling and clear Environment on Jumps;
- PS17 - Implement a recursive call processing and a few more bytecodes cleaning up the Environment - total score recovery of 2.5% since PS13;
- PS18 - Rebase on master which introduces a 4% regression on Octane due to removed optimization;
- PS19 - Fix off-by-one errors and clear Environment on context-related bytecodes - this reduces mjsunit test failures to only 1;
- PS20 - Fix more off-by-one errors so that now mjsunit passes successfully;
- PS29 - Switching loads to collect maps instead of values feedback has resulted in another 5% overall regression;
- PS30-PS36 - Mostly non-functional review fixes, no change in the performance either;

A set of smaller CLs that implement the basic functionality:

- https://chromium-review.googlesource.com/c/v8/v8/+/1386872

Appendix

The implementation can be found here.

[UPDATE] The declaration and usage example below have changed after reiterating on the implementation, please check the actual source code for an up-to-date version.

Declaration of the Environment class

```
class Environment : public ZoneObject {
 public:
  // Constructor from BytecodeArrayAccessor, JSFunction (the closure),
  // SFI, feedback

  // Transfers the registers state (the inferred_state vector) to the new
  // environment parameters. Called by Call* visitors.
  Environment* NewFromCall(InferredState receiver_state, InferredState
callee_state);

  // Will use the merge machinery for simulating control flow.
```

```
  void Merge(Environment* other);

  // Additional API types used below:
  typedef ZoneVector<Handle<Map>> MapVector;
  typedef ZoneVector<Handle<Object>> ValueVector;
  struct InferredState {
    enum State { None, Maps, ValuesAndMaps };
    MapVector inferred_maps;
    ValueVector inferred_values;
  };

  // Getters for the inferred state.
  // Used by Sta* visitors.
  InferredState LookupAccumulator() const;
  InferredState LookupRegister(Register the_register) const;

  // Setters for the internal machine state.
  // Used by Lda* visitors.

  // Will add the constant value to the InferredState and set its type to
"ValuesAndMaps".
  void AddSeenAccumulatorValue(Handle<Object> constant);
  // Will add the map to the InferredState and set its type to "Maps".
  void AddSeenAccumulatorMap(Handle<Map> map);
  // Will set the InferredState to "None".
  void ClearAccumulator();
  // Replace the accumulator state completely
  void SetAccumulatorInferredState(InferredState new_state);

  // Same as the 4 above, but for the registers.
  void AddSeenRegisterValue(Register the_register, Handle<Object> constant);
  void AddSeenRegisterMap(Register the_register, Handle<Map> map);
  void ClearRegister(Register the_register);
  void SetRegisterInferredState(Register the_register, InferredState
new_state);
```

```
 private:
  // Contains the currently known state of the registers, the accumulator
  // and the parameters. The structure looks like:
  // values: parameters | registers | accumulator
  // indices:         reg_base   acc_base
  ZoneVector<InferredState> inferred_state;
  int reg_base;
  int acc_base;

  Handle<SharedFunctionInfo> sfi_;
  Handle<FeedbackVector> fv_;
}
```

Usage example

```
// The JavaScript source:
function foo() {
  return 42;
}

function bar() {
  foo();
}

// ------------------------------------

// bar generates the following bytecode:
LdaGlobal [0], [0]
Star r0
CallUndefinedReceiver0 r0, [2]

// ------------------------------------

// which eventually the visitors turn into a call to:
TraverseBytecode(env, bytecode_iterator) {
```

```
  // LdaGlobal [0], [0]
  name = get from bytecode_iterator
  if (value of name known at compile time) {
    value = get from heap(name)
    env->AddSeenAccumulatorValue(value)
  }

  // Star r0
  register = get from bytecode_iterator
  state = env->LookupAccumulator()
  env->SetRegisterInferredState(register, state)

  // CallUndefinedReceiver0 r0, [2]
  callee_reg = get from bytecode_iterator
  callee = env->LookupRegister(callee_reg)
  receiver_reg = get from bytecode_iterator
  receiver = env->LookupRegister(receiver_reg)

  new_env = env->NewFromCall(receiver, callee)
  new_bytecode_iterator = get from callee

  // Recurse
  TraverseBytecode(new_env, new_bytecode_iterator)
}
```