Ignition: Register Equivalence Optimization

Author: oth@chromium.org
Last Updated: 2016/04/11

This note describes an online optimization technique to reduce the number of register transfers in bytecode emitted by the Ignition bytecode generator. The reduction in register transfers translates into fewer instruction dispatches, lower memory consumption, and potentially smaller bytecode frames.

Introduction

The V8 Ignition interpreter implements a register machine plus accumulator model. The accumulator is a non-user visible register that is used to hold intermediate results. Function arguments and local variables are held in user visible registers and non-user visible temporary registers are emitted to hold intermediate results in expression stacks. Temporary registers also find uses in other places such as managing non-programmer visible state for constructs like for..in.

The current bytecode generator takes care to avoid assignment hazards (see appendix) and invariably emits many pairs of move-register-to-accumulator, move-accumulator-to-register that could be more compactly expressed as single move-register-to-register bytecodes. The bytecode generator also creates temporaries that shadow locals and these temporaries appear as operands to bytecodes. In many cases the uses of the temporary can be avoided and the register corresponding to the local be used directly as an operand. The barrier to these improvements today is that neither the bytecode generator nor the bytecode array builder has sufficient visibility into the bytecode stream.

Recent changes to the Ignition source code have annotated register operands to mark whether they are used for inputs and outputs and similarly whether the accumulator is read or written by a bytecode. We use this information together in a new component inserted after the bytecode array builder to optimize register transfers with an online algorithm.The problem is strictly simpler than traditional register allocation and register assignment, but still requires care when dealing with basic block boundaries and preserving offsets for branches and source code positions. We have built a prototype implementation of the ideas presented here and it has no significant test regressions (11 failures relating to source positions in mjsunit[1]).

[1] Currently looking for a solution to these failures.

Register Equivalence

The core idea is to defer register transfers to and from registers that are not user observable until it is established that they are required. For debugging purposes, registers corresponding to local variables and parameters must always be observably correct. However, the accumulator and temporary registers emitted by the bytecode generator are not visible to the user, or the debugger, and can be deferred or neglected if not required.

We define a register equivalence set as a set of registers (including the accumulator) that have an equivalent value by virtue of one register being transferred to another. Members of the set are said to be materialized if they can be seen to currently hold the value in the output bytecode stream. Register transfer operations (`Ldar`, `Star`, `Mov`) have the effect of adding registers to equivalence sets and moving registers between equivalence sets. These operations will sometimes result in register transfers being emitted. Other bytecodes are evaluated for their register dependencies and registers are materialized to meet their requirements. When registers are in equivalent sets they can be potentially substituted in the operands associated with a bytecode. The side-effects of these bytecodes on register state are also evaluated to update the register equivalence sets.

As an example, consider the following trivial function:

```
function demo(x, y) {
  return Math.pow(x, y) + x * y;
}
```

The Ignition bytecode generator produces the code on the left hand side in below which is refined by the register equivalence optimization (REO) to produce the code on the right hand side.

	BytecodeGenerator Output	**REO Equivalent Output**
0	`StackCheck`	`StackCheck`
	`// 0. Load the global Math object`	`// 0. Load the global Math object`
1	`LdaGlobal [0], [3]`	`LdaGlobal [0], [3]`
2	`Star r2`	`Star r2`
	`// 1. Load the pow descriptor`	`// 1. Load the pow descriptor`
3	`LoadIC r2, [1], [5]`	`LoadIC r2, [1], [5]`
4	`Star r1`	`Star r1`
	`// 2. Prepare registers for Call`	`// 2. Prepare registers for Call`
5	`Ldar a0`	`Mov a0, r3`
6	`Star r3`	`Mov a1, r4`

7	`Ldar a1`	
8	`Star r4`	
	`// 3. Call pow(x, y)`	`// 3. Call pow(x, y)`
9	`Call r1, r2, #3, [1]`	`Call r1, r2, #3, [1]`
10	`Star r0`	`Star r0`
	`// 4. Calculate x * y`	`// 4. Calculate x * y`
11	`Ldar a0`	
12	`Star r1`	
13	`Ldar a1`	`Ldar a1`
14	`Mul r1`	`Mul a0`
15	`Add r0`	`Add r0`
16	`Return`	`Return`

The register equivalence optimizer reads bytecode generator output and proceeds as shown below. The bars above registers indicate that a register is materialized at this point when evaluating the code from the bytecode generator, no bar indicates the register is not materialized.

Bytecode Generator Output	Equivalence Sets
Line 1 materializes the accumulator and puts it in a new equivalence set.	$\{\overline{A}\}$
Line 2 places `r2` into the same equivalence set, but no store is emitted yet.	$\{\overline{A}, r_2\}$
Line 3 the `LoadIC` will clobber the accumulator and `r2` is required as an operand. Register `r2` is materialized storing the accumulator. The accumulator is placed in a new equivalence set.	$\{\overline{A}\}\{\overline{r_2}\}$
Line 4 adds `r1` to accumulator's equivalence set, no store is emitted yet.	$\{\overline{A}, r_1\}\{\overline{r_2}\}$
Lines 5...8 establish further equivalences. `r1` must be materialized before the accumulator is loaded in line 5, causing `Star r1` to be emitted.	$\{\overline{r_1}\}\{\overline{a_0}, r_3\}$ $\{\overline{a_1}, r_4, \overline{A}\}$
Line 9 has a `call` bytecode requiring registers `r1` to `r4` which need to be materialized before `call` is emitted. Accumulator moved to a new set as it holds the result from `call`.	$\{\overline{r_1}\}\{\overline{a_0}, \overline{r_3}\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{A}\}$
Line 10 places `r0` into the accumulator's equivalence set.	$\{\overline{r_1}\}\{\overline{a_0}, \overline{r_3}\}$ $\{\overline{a_1}, \overline{r_4}\}\{r_0, \overline{A}\}$
Line 11 moves the accumulator into `a0`'s equivalence set. Because `r0` in the old equivalence set is not materialized, it must be now.	$\{\overline{r_1}\}\{\overline{a_0}, \overline{r_3}, A\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{r_0}\}$

Line 12 places `r1` into the accumulators equivalence set.	$\{\overline{a_0}, \overline{r_3}, A, r_1\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{r_0}\}$
Line 13 establishes an equivalence between the accumulator and `a1`.	$\{\overline{a_0}, \overline{r_3}, r_1\}$ $\{\overline{a_1}, \overline{r_4}, A\}\{\overline{r_0}\}$
Line 14 is the multiply instruction which requires the accumulator and `r1` as inputs. However, `r1` is equivalent to `a0` and `a0` is already materialized. The `mul` bytecode places the accumulator in a new equivalence set.	$\{\overline{a_0}, \overline{r_3}, r_1\}\{A\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{r_0}\}$
Line 15 adds the saved result from the calculation of `Math.pow(x,y)`.	$\{\overline{a_0}, \overline{r_3}, r_1\}\{A\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{r_0}\}$
Line 16 returns from the function.	$\{\overline{a_0}, \overline{r_3}, r_1\}\{A\}$ $\{\overline{a_1}, \overline{r_4}\}\{\overline{r_0}\}$

In the prototype implementation, the bytecode generator is able to provide the register equivalence optimizer with details of when temporary registers die. This enables temporary registers to be pruned from equivalence sets when they cease to exist. The bytecode generator uses scopes to allocate temporary registers, and the temporaries used to ensure a continuous range of registers for `call` on line 9 are dead by line 11.

Basic Block Boundaries

Equivalence relations are broken at basic block boundaries. All live registers are materialized into their own sets before transitioning to a new basic block. The scoped register allocation in the bytecode generator means that most temporaries do not need to be materialized, only those that are live. Live temporaries can legitimately exist at basic block boundaries, for example the pattern for emitting `for..in` loops uses three temporary variables for holding state and these are live in the outer scope of the `for..in` loop. Other cases such as logical conditions within expressions can result in temporaries kept alive across a basic block boundary as the basic block is introduced into the scope of the expression evaluation. The majority of temporary variables are used in expressions or just for a specific statement and these die before basic block boundaries are reached.

The implementation gets a notification from the bytecode array builder when a new basic block is about to be transitioned to at the points immediately before a label is bound and immediately before a jump is emitted. The implementation then ensures all live registers including the accumulator are materialized and placed into their own unique equivalent sets at these boundaries.

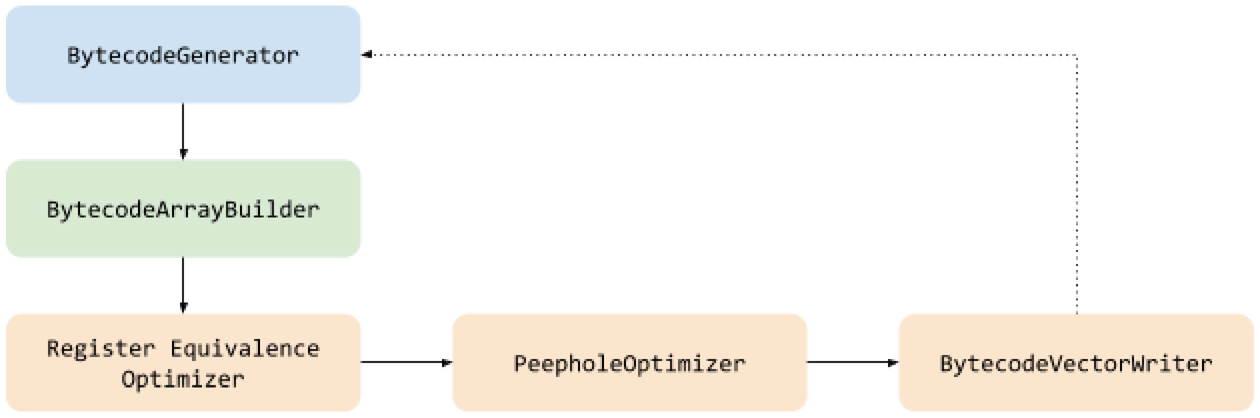
As code in the basic block is visited, new equivalence relations are established and the optimization process is applied to the basic block.

Prototype Implementation

A prototype implementation of the register equivalence optimizer exists. It is plumbed into the bytecode generation process after the bytecode array builder which the bytecode generator uses to emit bytecodes. The optimizer requires approximately 500 lines of C++ code to implement.

The existing bytecode peephole optimizer in the bytecode array builder needed to be moved after the REO optimizer as the peephole optimizer peeks at the bytecode output which is not necessarily consistent when the peephole optimizer peeks due to the suppression of loads and stores. Additionally, the existing peephole optimizations apply to some patterns of loads and stores which the peephole optimizer would not observe it's current location (embedded within the bytecode array builder).

In the prototype, the bytecode generation pipeline looks like:



The register equivalence optimizer, peephole optimizer, and bytecode vector writer support a common `BytecodeWriter` interface. This allows each stage to tested independently and keeps the functionality isolated from other components.

The bytecode writer interface supports pushing a bytecode into the pipeline via a `Write()` method. It also supports two kinds of flushes to clear the pipeline. Flushes are needed when the bytecode array builder needs to ascertain an offset into the bytecode, either for computing jump offsets (also exception offset) or offsets for source position information. The corresponding flush methods are `FlushForBranch()` and `FlushForSourcePosition()`. Each pipeline element performs the flush locally and then calls downstream to continue the flushing operation.

`FlushForBranch()` materializes unmaterialized registers and breaks equivalence sets assuming a branch is about to be emitted in the register equivalence optimizer. It pushes loads and stores down the pipeline and these are flushed out of the peephole optimizer. `FlushForSourcePosition()` has no effect on the register equivalence optimizer, but causes the peephole optimizer to flush any buffered instructions to ensure that the source position is marked at the correct location.

The BytecodeWriter interface also has a LeaveBasicBlock() method that causes the peephole optimizer to push buffered output downstream and set its internal buffer size to zero.

The new peephole optimizer may hold onto history of instructions in the basic block even when they've been flushed. It only does this when the flush does not correspond to a basic block boundary. The history is kept because some peephole optimizations can be applied when looking at the last instruction flushed in combination with the next instruction pushed down the pipeline. For instance, the bytecode generation process passes a `JumpIfToBooleanTrue` to the peephole optimizer, then this can be specialized to a plain `JumpIfTrue` if the previous instruction is known to load the accumulator with a boolean value, e.g. a test instruction or boolean load. If a flush occurs between those instructions, then the current `JumpIfToBooleanTrue` instruction can still be specialized.

Prototype Performance

The quantity of bytecodes emitted and their size is a function of the code fed to the bytecode generator. During the development of the prototype, we have been focussing on the Octane 2.1 benchmark. The size reductions observed are shown in the table below.

	Total Code Size without REO, bytes	Total Code Size with REO, bytes	Reduction
Octane 2.1	38,624,976	28,967,224	25%
Octane 2.1 without zlib and typescript	14,198,664	13,361,792	6%

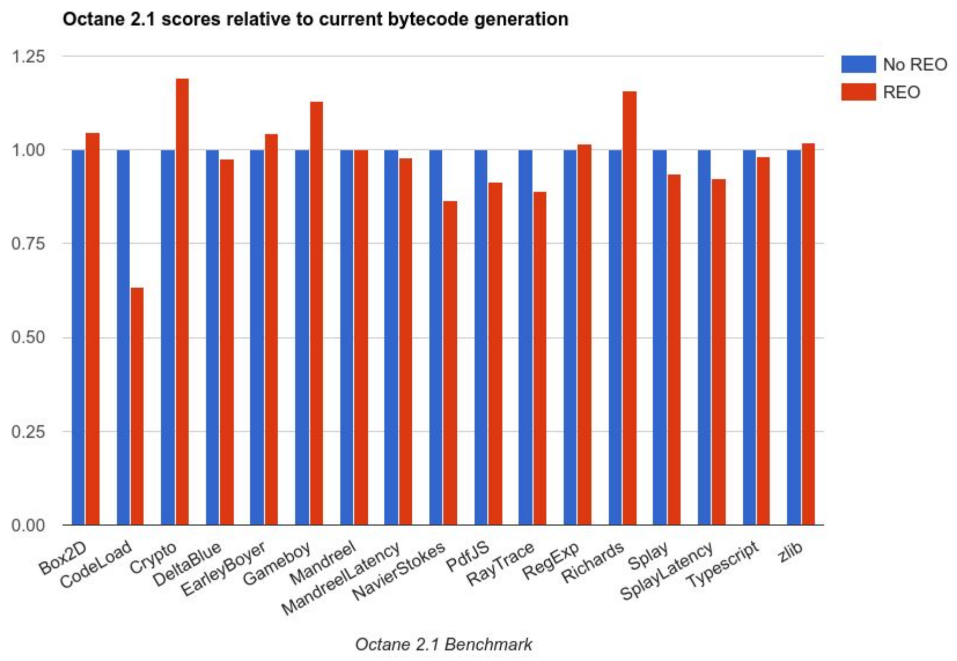
Code sizes for Octane run with Ignition enabled, compilers disabled, and minimum iterations

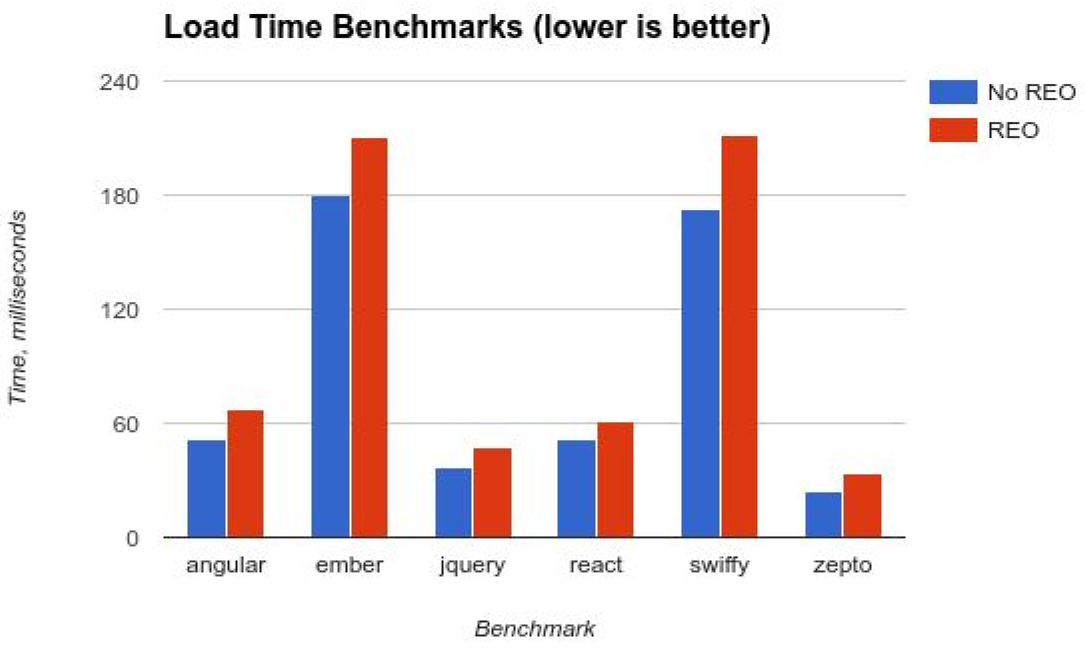
	Elapsed execution time without REO, seconds	Elapsed execution time with REO, seconds	Reduction
Octane 2.1	455	408	10%
Octane 2.1 without zlib and typescript	77	69	10%

Elapsed execution wall times for Octane run with Ignition enabled,compilers disabled, and minimum iterations. Running x64 build on Intel Xeon E5-2690 @ 2.9GHz

In these measurements, zlib takes by far the longest time to run, and typescript causes the most code to be generated. Ignition is compiling bytecode eagerly at this point in the tree and this may account for the amount of code generated for the typescript benchmark.

Although, REO reduces the total running time of Octane, the average change in scores across all components is effectively zero. The results we have so far are shown in the next figure. The scores are down for roughly 50% of benchmarks. We're in the process of examining the worst cases to see if there are mitigations to improve this.


Octane 2.1 scores relative to current bytecode generation
No REO
REO
1.25
1.00
0.75
0.50
0.25
0.00
Box2D
CodeLoad
Crypto
DeltaBlue
EarleyBoyer
Gameboy
Mandreel
MandreelLatency
NavierStokes
PdfJS
RayTrace
RegExp
Richards
Splay
SplayLatency
Typescript
zlib
Octane 2.1 Benchmark


Load Time Benchmarks (lower is better)
No REO
REO
240
180
120
60
0
Time, milliseconds
angular
ember
jquery
react
swiffy
zepto
Benchmark

Discussion

REO started as an attempt to avoid the complexity of assignment hazard avoidance in the bytecode graph generator. The bytecode generator is a complex component in the interpreter translating the AST into bytecode and dealing with control flow, exceptions, and all language constructs exposed in the AST. Being overly concerned with assignment hazard collisions in the bytecode generator is a path to indecipherable complexity. The REO optimizer is completely distinct from the bytecode generator and the bytecode array builder. This makes it independently testable and optionally easy to turn on and off.

REO improves our register usage in many scenarios, by replacing sequences of `Ldar;Star` with `Mov`’s saving memory and reducing the number of bytecodes to dispatch.

The current prototype has moved the peephole optimizer downstream and made it distinct from the bytecode array builder. This is essential for the peephole optimizer not to interfere with the behaviour of REO and is better for testing.

The implementation details of the prototype are not discussed in detail in this document. We thought hard about how to implement equivalence sets efficiently and place the set information for each register into linked lists. Nodes in the list contain the register, one bit indicating whether the register is materialized, and a sequence number for selecting the best materialized alternative for temporaries. The implementation uses a zone to allocate nodes, but keeps them in a free list when registers die to avoid excessive allocation load on large code bodies.

Future Opportunities

Frame size reduction - currently the bytecode array builder computes the bytecode frame sizes based on temporary register usage. REO means there may be fewer temporaries present in the output bytecode.

Peephole optimizer has a better view of previous bytecodes now. It is well suited for replacing bytecode sequences with super-bytecodes for sequences that we find through perf analysis and can be implemented maintaining deoptimization requirements.

Potential peephole optimization of `LoadIc;Star` and `LdaGlobal;Star` to reduce number of dispatched bytecodes. This pattern seems recurrent in benchmarks.

Peephole optimization of constant loads to registers, e.g. replace `LdaSmi;Star` with a new bytecode `LdrSmi`. Similarly with `LdaConstant;Star` being interchangeable with `LdrConstant`.

Peephole optimization of `LdaZero;Star r1;LdaZero;Star r2` to remove second and subsequent LdaZero's. This could also be done by treating LdaZero as loading a 'magic' register that the accumulator has equivalence with. We have a gap between parameters and locals/temporaries in the register space due to frame layout.

Materialization at basic block boundaries will on some occasions materialize temporaries that are not used in successor blocks. An optional second optimizer could do a pass to eliminate these and also compact gaps in the temporary register space remaining after REO has eliminated temporaries.

REO preserves updates to user visible registers primarily so debugging is correct. More registers and computationally unnecessary loads and stores could be removed more aggressively if it were known the debugger was not attached. Having different code for debug and optimized execution is likely unattractive, but is technically feasible.

Address source position issue in the 11 mjsunit test failures.

Appendix I - Assignment Hazard

Assignment hazards occur for register machines when an intermediate result in an expression might be modified by evaluating a term further along an expression. They do not occur in stack machines as the intermediate results are pushed onto the stack and unavailable. An example expression that has a potential hazard is:

```
      var x = 10;
      var y = x + (x = 3);
```

In the assignment of y involves the evaluation of a binary operator. The left-hand side is evaluated first and then the right hand side modifies the term used in the left-hand side. The simple solution for this is to assign the left hand to a temporary register when it is evaluated and then evaluate the right hand side. The temporary register is only needed when subsequent terms in the expression are evaluated and conflict with earlier terms. In cases where there is no conflict, the generator emits unnecessary moves to temporaries.

We have attempted to avoid assignment hazards in three different ways:

1) Two passes. The bytecode generator aggressively resolves terms to registers and has a conflict set that it maintains whilst evaluating the expression. The conflict set contains all registers read during the expression evaluation and a hazard occurs if a write occurs to any of these registers during evaluation. If a conflict is discovered, a second pass is made when the expression has been completely evaluated. On the second pass,

temporaries are used for registers with conflicts. We didn't like the two pass nature of this approach nor different handling of registers on the two passes.

2) Using temporaries for terms on the right-hand side when writes are discovered. This technique maintained the conflict set and dynamically re-mapped register writes on the rhs to temporaries and then updated the register values after the expression is evaluated. This failed to deal with cases that involved named and keyed load and store operations.
3) A ticket based scheme that led to allocating temporaries for the intermediate values on the left-hand-side when a conflict is detected. It currently fails for similar reasons to 2 and is similarly hard to reason about.

All three of these techniques add considerable complexity to the already complex bytecode generator. They are hard to reason about and to test. Our current bytecode generator implementation takes the most conservative approach of always allocating temporaries for intermediate expression values. It is easy to reason about and results in the correct execution of the v8 test suites.
Here's a more subtle assignment hazard example (mjsunit/regress/regress-286.js):

```
function test() {
  var o = [1];
  var a = o[o ^= 1];
  return a;
};
```

which returns the value 1. This function translates to the following bytecodes:

	BytecodeGenerator Output	**REO Equivalent Output**
0	`StackCheck`	`StackCheck`
1	`CreateArrayLiteral [0], [0], #3`	`CreateArrayLiteral [0], [0], #3`
2	`Star r0`	`Star r0`
3	`Star r2`	
4	`Ldar r0`	
5	`Star r3`	
6	`LdaSmi [1]`	`LdaSmi [1]`
7	`BitwiseXor r3`	`BitwiseXor r0`
8		`Mov r0, r2`
9	`Star r0`	`Star r0`
10	`KeyedLoadIC r2, [1]`	`KeyedLoadIC r2, [1]`
11	`Star r1`	`Star r1`
12	`Return`	`Return`

We found this example hard to address cleanly in the bytecode generator due to the nesting of AST visitor methods. The bytecode generator emits a correct solution using two temporaries. The REO equivalent maintains the same correctness, but with one temporary and 3 few bytecodes.