BRANCH HINTING UPDATE

PROPOSAL SUMMARY

Improve the performance of machine code generated by Wasm engines, by hinting that a particular conditional branch destination is very likely/unlikely.

This allows the engine to make better decisions for code layout (improving instruction cache hits) and register allocation.

The hints are listed in a custom section (`metadata.code.branch_hint`).

BINARY FORMAT:

```
branchhintsec     ::= section_0(branchhintdata)
branchhintdata    ::= n:name                                  (if n = 'metadata.code.branch_hint')
                      vec(funcbranchhints)
funcbranchhints   ::= fidx:funcidx vec(branchhint)
branchhint        ::= instoff:u32 l:u32 branchhintkind
branchhintkind    ::= 0x00
                    | 0x01
```

TEXT FORMAT:

```
branchhintannot  ::= '(@metadata.code.branch_hint' branchhintstr ')'
branchhintstr    ::= '''\00'''
                   | '''\01'''
```

EXAMPLE USAGE (TEXT FORMAT)

```
(func $test1 (type 0)
  (local i32)
  local.get 1
  local.get 0
  i32.eq
  (@metadata.code.branch_hint "\00" ) if
    call $foo
    return
  end
  return
)
```

CUSTOM SECTIONS/ANNOTATIONS WORK

Support for testing custom sections and annotations added in the reference interpreter and test suite.

Testing includes checking for text/binary format syntax and semantics whithin the module (e.g. make sure that a branch hint is attached to a branch instruction), plus round tripping check.

Other users of the test suite can implement `assert_invalid_custom`/`assert_malformed_custom` if they want.

CUSTOM HANDLER INTERFACE

```
type custom = custom' Source.phrase
and custom' =
{
  name : Ast.name;
  content : string;
  place : place;
}
...
module type Handler =
sig
  type format'
  type format = format' Source.phrase
  val name : Ast.name
  val place : format -> place
  val decode : Ast.module_ -> string -> custom -> format (* raise Code *)
  val encode : Ast.module_ -> string -> format -> custom
  val parse : Ast.module_ -> string -> Annot.annot list -> format list (* raise
  val arrange : Ast.module_ -> Sexpr.sexpr -> format -> Sexpr.sexpr
  val check : Ast.module_ -> format -> unit (* raise Invalid *)
end
```

Run the interpreter with:

```
./wasm -c metadata.code.branch_hint -c name -c <myhandler> -t test.wast
```

PHASE STATUS

Currently phase 3.

Requirements for phase 4:

- Two or more Web VMs implement the feature ✅ (V8, JSC)
- At least one toolchain implements the feature ✅ (Cheerp)
- The formalization and the reference interpreter are usually updated
  - formalization ✅
  - reference interpreter ❌ -> ✅
- Community Group has reached consensus in support of the feature ❔

Leftover from phase 3 requirements:

-Test suite has been updated to cover the feature in its forked repo ❌ -> ✅

MISSING PIECES

- Custom annotations phase 4
  - same blockers as Branch Hinting. Now solved.
  - Latest Pr #19 adds new assertions kinds for custom sections to the interpreter
  - Minor issue: how to test changes in test/harness/ ?
- Other hints, e.g. "no-inline"? Issue #18
  - Add other kinds of hints to this section, or define new ones

EXTRA - V8 IMPLEMENTATION

br_on_null

```
void WasmGraphBuilder::BrOnNull(Node* ref_object, Node** null_node,
                                Node** non_null_node) {
  BranchExpectFalse(gasm_->WordEqual(ref_object, RefNull()), null_node,
                    non_null_node);
}
```

br_if

```
  void BrIf(FullDecoder* decoder, const Value& cond, uint32_t depth) {
    SsaEnv* fenv = ssa_env_;
    SsaEnv* tenv = Split(decoder->zone(), fenv);
    fenv->SetNotMerged();
    WasmBranchHint hint = WasmBranchHint::kNoHint;
    if (branch_hints_) {
      hint = branch_hints_->GetHintFor(decoder->pc_relative_offset());
    }
    switch (hint) {
      case WasmBranchHint::kNoHint:
        builder_->BranchNoHint(cond.node, &tenv->control, &fenv->control);
        break;
      case WasmBranchHint::kUnlikely:
        builder_->BranchExpectFalse(cond.node, &tenv->control, &fenv->control);
        break;
      case WasmBranchHint::kLikely:
        builder_->BranchExpectTrue(cond.node, &tenv->control, &fenv->control);
        break;
    }
    builder_->SetControl(fenv->control);
    SetEnv(tenv);
    BrOrRet(decoder, depth, 1);
```

EXTRA - BENCHMARKS

Measured different workloads running in CheerpX (X86 VM and JIT compiler). CheerpX generate branch hints in JITted modules when checking for slow paths.

Average speedup: 7-10%.

The average C/C++ program will likely not benefit as much, but interpreters, runtimes, VMs, JITs, ... will.

Benchmark environment available here:
https://yuri91.github.io/webvm-benches