JIT Compilation

David Chisnall
University of Cambridge
LLVM Summer School, Paris, June 13, 2017

Late Binding

- Static dispatch (e.g. C function calls) are jumps to specific addresses
- Object-oriented languages decouple method name from method address
- One name can map to multiple implementations (e.g. different methods for subclasses)
- Destination must be computed somehow

Example: C++

- Mostly static language
- Methods tied to class hierarchy
- Multiple inheritance can combine class hierarchies

```cpp
class Cls {
    virtual void method();
};
// object is an instance of Cls or a subclass of Cls
void function(Cls *object) {
    // Will call Cls::method or a subclass override
    object->method();
}
```

Example: JavaScript

- Prototype-based dynamic object-oriented language
- Objects inherit from other objects (no classes)
- Duck typing

```
a.method = function() { ... };
...
// Will call method if b or an object on
// b's prototype chain provides it.  No
// difference between methods and
// instance/ variables: methods are just
// instance variables containing
// closures.
b.method();
```

VTable-based Dispatch

- Tied to class (or interface) hierarchy
- Array of pointers (virtual function table) for method dispatch
- Method name mapped to vtable offset

```
struct Foo {
    int x;
    virtual void foo();
};
void Foo::foo() {}

void callVirtual(Foo &f) {
    f.foo();
}
void create() {
    Foo f;
    callVirtual(f);
}
```

Calling the method via the vtable

```
define void @_Z11callVirtualR3Foo(%struct.Foo* %
    f) uwtable ssp {
  %1 = bitcast %struct.Foo* %f to void (%struct.
      Foo*)***
  %2 = load void (%struct.Foo*)*** %1, align 8,
      !tbaa !0
  %3 = load void (%struct.Foo*)** %2, align 8
  tail call void %3(%struct.Foo* %f)
  ret void
}
```

Call method at index 0 in vtable.

Creating the object

```
@_ZTV3Foo = unnamed_addr constant [3 x i8*] [
  i8* null,
  i8* bitcast ({ i8*, i8* }* @_ZTI3Foo to i8*),
  i8* bitcast (void (%struct.Foo*)*
     @_ZN3Foo3fooEv to i8*)]

define linkonce_odr void @_ZN3FooC2Ev(%struct.
   Foo* nocapture %this)  {
  %1 = getelementptr inbounds %struct.Foo* %this
     , i64 0, i32 0
  store i32 (...)** bitcast
    (i8** getelementptr inbounds ([3 x i8*]*
       @_ZTV3Foo, i64 0, i64 2) to i32 (...)**),
    i32 (...)*** %1
}
```

Devirtualisation

- Any indirect call prevents inlining
- Inlining exposes a lot of later optimisations
- If we can prove that there is only one possible callee, we can inline.
- Easy to do in JIT environments where you can *deoptimise* if you got it wrong.
- Hard to do in static compilation

Problems with VTable-based Dispatch

- VTable layout is per-class
- Languages with duck typing (e.g. JavaScript, Python, Objective-C) do not tie dispatch to the class hierarchy
- Dynamic languages allow methods to be added / removed dynamically
- Selectors must be more abstract than vtable offsets (e.g. globally unique integers for method names)

Lookup Caching

- Method lookup can be slow or use a lot of memory (data cache)
- Caching lookups can give a performance boost
- Most object-oriented languages have a small number of classes used per callsite
- Have a per-callsite cache

Callsite Categorisation

- Monomorphic: Only one method ever called
  - Huge benefit from inline caching
- Polymorphic: A small number of methods called
  - Can benefit from simple inline caching, depending on pattern
  - Polymorphic inline caching (if sufficiently cheap) helps
- Megamorphic: Lots of different methods called
  - Cache usually slows things down

Inline caching in JITs

- Cache target can be inserted into the instruction stream
- JIT is responsible for invalidation
- Can require *deoptimisation* if a function containing the cache is on the stack

Speculative inlining

- Lookup caching requires a mechanism to check that the lookup is still valid.
- Why not inline the expected implementation, protected by the same check?
- Essential for languages like JavaScript (lots of small methods, expensive lookups)

Inline caching

```
kup_fn
```

```
, $last, fail
hod
:
```

- First call to the lookup rewrites the instruction stream
- Check jumps to code that rewrites it back

Polymorphic inline caching

```
, $expected, cls
hod



, $expected2, cls
hod
```

- Branch to a jump table
- Jump table has a sequence of tests and calls
- Jump table must grow
- Too many cases can offset the speedup

Trace-based optimisation

- Branching is expensive
- Dynamic programming languages have lots of method calls
- Common hot code paths follow a single path
- Chain together basic blocks from different methods into a trace
- Compile with only branches leaving
- Contrast: trace vs basic block (single entry point in both, multiple exit points in a trace)

Type specialisation

- Code paths can be optimised for specific types
- For example, elide dynamic lookup
- Common case: `a+b` is much faster if you know `a` and `b` are integers!
- Can use static hints, works best with dynamic profiling
- Must have fallback for when wrong

Deoptimisation

- Disassemble existing stack frame and continue in interpreter / new JIT'd code
- Stack maps allow mapping from register / stack values to IR values
- Fall back to interpreter for new control flow
- NOPs provide places to insert new instructions
- New code paths can be created on demand
- Can be used when caches are invalidated or the first time that a cold code path is used

LLVM: Anycall calling convention

- Used for deoptimisation
- All arguments go somewhere
- Metadata emitted to find where
- Very slow when the call is made, but no impact on register allocation
- Call is a single jump instruction, small instruction cache footprint
- Designed for slow paths, attempts not to impact fast path

Deoptimisation example

JavaScript:

```
c;
```

Deoptimisable pseudocode:

```
if (!(is_integer(b) && is_integer(c)))
    anycall_interpreter(&a, b, c); // Function
        does not return
a = b+c;
```

Case Study: JavaScriptCore (WebKit)

- Production JavaScript environment
- Multiple compilers!

JavaScript is odd

- Only one numeric type (double-precision floating point)
- Purely imperative - no declarative class structures
- No fixed object layouts
- Code executes as loaded, must start running before download finishes
- Little scoping

Web browsers are difficult environments

- Most JavaScript code is very simple
- Fast loading is very important
- Some JavaScript is very CPU-intensive
- Fast execution is important
- Users care a lot about memory usage!

Before execution

- JSC reads code, produces AST, generates bytecode
- Bytecode is dense and the stable interface between all tiers in the pipeline

Contrast: V8

- Initial parse skips text between braces
- No stored IR, AST (just pointers into the code)
- Recompilation includes reparse of relevant parts

Overall design: multiple tiers

- First tiers must start executing quickly
- Hot code paths sent to next tiers
- Last tier must generate fast code

Compare with simplified MysoreScript: Two tiers (AST interpreter / JIT), functions promoted to JIT after 10 executions.

First tier: LLInt, a bytecode interpreter

- Very fast to load
- Written in custom low-level portable assembly
- Simple mapping from each asm statement to host instruction
- Precise control of stack layout, no C++ code
- 14KB binary size: fits in L1 cache!

Second tier: Baseline JIT

- LLInt reads each bytecode, dispatches on demand
- After 6 function entries or 100 statement invocations, JIT is triggered
- Simple bytecode JIT, pastes asm similar to LLInt into sequences.
- Exactly the same stack layout as LLInt.
- Introduces polymorphic inline caching for heap accesses
- Works at method granularity

Why is stack layout important?

- Partial traces may be JIT'd
- Must be able to jump back to LLInt for cold paths
- Remember: Deoptimization

Type feedback

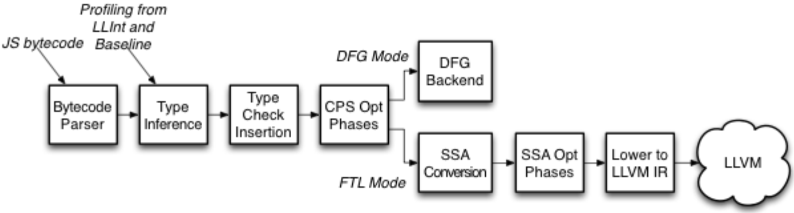
- Pioneered by Self
- Both LLInt and the baseline JIT collect type information
- Later tiers can optimise based on this
- More useful than type inference for optimisation (this is usually type X, vs this type must always be type X, Y, or Z)

General note: for optimisation, X is usually true is often more helpful than Y is always true if X is a much stronger constraint than Y (and X is cheap to check).

Other profiling

- Function entry
- Branch targets
- Build common control flow graphs

Tiers 3/4: the high-performance JITs



LLVM usage now replaced by B3 (Bare Bones Backend). LLV8 still uses LLVM for a last-tier JIT in V8.

CPS Optimisers

- Continuation-passing style IR
- Every call is a tail call, all data flow is explicit
- Lots of JavaScript-specific optimisations
- Many related to applying type information
- CPS not covered much in this course, but lots of recent research on combining the best aspects of SSA and CPS!

Type inference

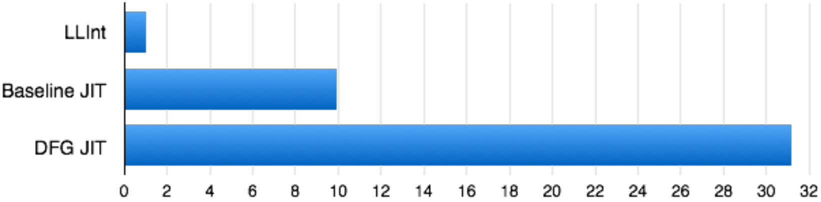
- Static type inference is really hard for dynamic languages
- Must be conservative: bad for optimisation
- Type feedback provided by earlier tiers
- Propagate forwards (e.g. int32 + int32 is probably int32: overflow unlikely)
- Fed back into later compiler stages
- LLInt and baseline JIT collect profiling information

Aside: Samsung's AoT JavaScript compiler

- Discontinued research project
- Used techniques from symbolic execution to statically find likely types for all code paths
- Generated optimised code
- Performance close to state-of-the-art JITs

Tier 3: Data flow graph JIT

- Speculatively inlines method calls
- Performs dataflow-based analyses and optimizations
- Costly to invoke, only done for hot paths
- Performs *on-stack replacement* to fall back to baseline JIT / LLInt



(higher is better)

Tier 4: LLVM / B3

- Input SSA is the output from the CPS optimisations
- Very high costs for optimisation
- Latency penalty avoided by doing LLVM compilation in a separate thread
- More advanced register allocator, low-level optimisations
- B3 does fewer optimisations, for lower latency (and power consumption), but still has much better register allocation than DFG JIT.

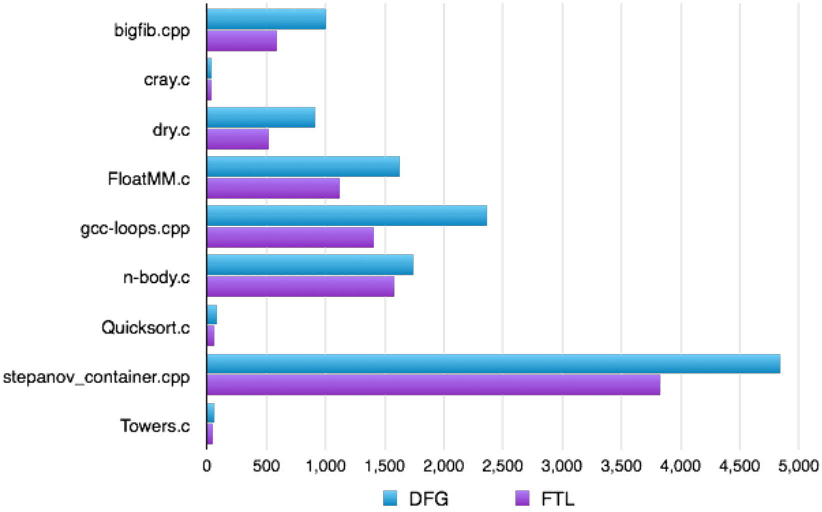
Patchpoints for deopimisation

- LLVM patchpoint provides jump to the runtime
- Stack map allows all live values to be identified
- Any that are needed for the interpreter are turned back into an interpreter stack frame
- Interpreter continues
- Deoptimisation means incorrect guesses in optimisation: fed back as profiling information

Patchpoints for object layout

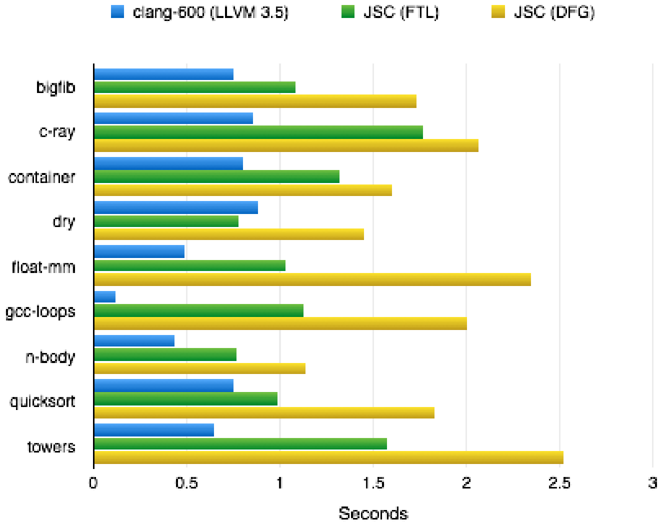
- Speculatively compiled assuming fixed field offsets
- Can become incorrect as more code is executed
- Dynamically patched with correct offsets when hit

FTL Performance (asm.js benchmarks)



(Lower is better)

FTL vs Clang



(Lower is better)

Lessons

- Modern compilers need a variety of different techniques
- There's no one-size-fits-all approach
- High-level transforms and microoptimisations are both needed
- JavaScript is designed to provide full employment for compiler writers
- JSC with FTL performance on asm.js code is similar to GCC from 10 years ago: there's no such thing as a slow language, only a slow compiler!

Lessons

- Modern compilers need a variety of different techniques
- There's no one-size-fits-all approach
- High-level transforms and microoptimisations are both needed
- JavaScript is designed to provide full employment for compiler writers
- JSC with FTL performance on asm.js code is similar to GCC from 10 years ago: there's no such thing as a slow language, only a slow compiler!

The End