Dynamic Dispatch and Duck Typing

L25: Modern Compiler Design

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
- Destination must be computed somehow

VTable-based Dispatch

- Tied to class (or interface) hierarchy
- Array of pointers (virtual function table) for method dispatch

```
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
}
```

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.

Problems with VTable-based Dispatch

- VTable layout is per-class
- Languages with duck typing do not tie dispatch to the class hierarchy
- Selectors must be more abstract than vtable offsets (e.g. globally unique integers for method names)

Ordered Dispatch Tables

- All methods for a specific class in a sorted list
- Binary (or linear) search for lookup
- Lots of conditional branches for binary search
- Either very big dtables or multiple searches to look at superclasses
- Cache friendly for small dtables (entire search is in cache)
- Expensive to add methods (requires lock / RCU)

Sparse Dispatch Tables

- Tree structure, 2-3 pointer accesses + offset calculations
- Fast if in cache
- Pointer chasing is suboptimal for superscalar chips (inherently serial)
- Copy-on-write tree nodes work well for inheritance, reduce memory pressure

Inverted Dispatch Tables

- Normal dispatch tables are a per-class (or per object) map from selector to method
- Inverted dispatch tables are a per-selector map from class (or object) to method
- If method overriding is rare, this provides smaller maps (but more of them)

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

Simple Cache

```
object.aMethod(foo);
```

```
static struct {
  Class cls;
  Method method;
} cache = {0, 0};
static Selector sel = compute_selector("aMethod"
   );
if (object->isa != cache->cls) {
  cache->cls = object->isa
  cache->method = method_lookup(cls, sel);
}
cache->method(object, sel, foo);
```

What's wrong with this approach?

Simple Cache

```
object.aMethod(foo);
```

```
static struct {
  Class cls;
  Method method;
} cache = {0, 0};
static Selector sel = compute_selector("aMethod"
    );
if (object->isa != cache->cls) {
  cache->cls = object->isa
  cache->method = method_lookup(cls, sel);
}
cache->method(object, sel, foo);
```

What's wrong with this approach? Updates? Thread-safety?

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

Inline caching

```
call lookup_fn
nop
```

```
bne $cls, $last, fail
call method
```

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

Variation: Guarded methods

- Specialised version of each method that knows the expected class
- Jump to the lookup function replaced by call to guarded method
- Method checks receiver type and tail-calls the lookup function if it's the wrong type

Polymorphic inline caching

```
bne $cls, $expected, cls
call method
ret
next:
bne $cls, $expected2, cls
call method
ret
```

- 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)

Prototype-based Languages

- Prototype-based languages (e.g. JavaScript) don't have classes
- Any object can have methods
- Caching per class is likely to hit a lot more cases than per object

Hidden Class Transforms

- Observation: Most objects don't have methods added to them after creation
- Create a hidden class for every constructor
- Also speed up property access by using the class contain fixed offsets for common properties

Type specialisation

- Code paths can be optimised for specific types
- For example, elide dynamic lookup
- 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

Deoptimisation example

JavaScript:

```
a = b + c;
```

Deoptimisable pseudocode:

```
if (!(is_integer(b) && is_integer(c)))
    anycall_interpreter(&a, b, c);
a = b+c;
```

Questions?