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
}
```

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 Inline Cache

```
[wobject aMethod:foo];
```

```
static struct {
  Class cls;
  Method method;
} cache = {0, 0};
static SEL 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 Inline Cache

```
[wobject aMethod:foo];
```

```
static struct {
  Class cls;
  Method method;
} cache = {0, 0};
static SEL 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?

Cache Scope

- Global cache

Cache Scope

- Global cache
  - Needs locking or lockless updates for multithreaded languages
  - False sharing problems

Cache Scope

- Global cache
  - Needs locking or lockless updates for multithreaded languages
  - False sharing problems
- Per-thread cache

Cache Scope

- Global cache
  - Needs locking or lockless updates for multithreaded languages
  - False sharing problems
- Per-thread cache
  - No Contention
  - Accessing TLS can be expensive in shared libraries

Cache Scope

- Global cache
  - Needs locking or lockless updates for multithreaded languages
  - False sharing problems
- Per-thread cache
  - No Contention
  - Accessing TLS can be expensive in shared libraries
- Method-Local Cache

Cache Scope

- Global cache
  - Needs locking or lockless updates for multithreaded languages
  - False sharing problems
- Per-thread cache
  - No Contention
  - Accessing TLS can be expensive in shared libraries
- Method-Local Cache
  - Allocated on stack
  - No synchronisation needed
  - Access is cheap
  - Cache only persists for current function duration

Statically Determining Cache Sites

- Policies from the GNUstep Objective-C Runtime
  - Superclass method invocation is always cached (rarely changes)
  - Class methods are always cached (rarely change)
  - Message sends in loops are cached with on-stack cache cheap cache checks
- Run-time type feedback (as in Self) can improve accuracy

Thread-Safe Inline Caching

- Method lookup returns a cacheable slot (structure) pointer
- Slot contains the method pointer and a version
- Cache contains slot pointer and a version
- Caches must be updated in a way that avoids data races

Thread-Safe Inline Caching Algorithm

```
static int cache_version;
static struct slot *cached_slot;
struct slot *slot = lookup_slot(cls, selector);
cache_version = 0;
slot->cached_for = cls;
cached_slot = slot;
cache_version = slot->version;
```

- Store 0 in version
- Store slot pointer in cache
- Store version from slot in cache
- All must be sequentially consistent atomic stores
- Slot and version either match, or version is 0

Thread-Safe Inline Caching Lookup

```
struct slot *slot = atomic_read(cached_slot);
if (slot->version == atomic_read(cached_version)
    &&
    slot->cached_for == cls->isa)
  slot->method(obj, sel);
}
```

- Read slot
- Read version
- Check class matches expected
- Call method

Inline Cache-Safe Method Update

- Replace method in slot
- Increment version for all slots with the same selector in subclasses
- No version increment for the same class (cached slot is still safe to use)

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

Decompilation

- Branches are expensive
- Code can be emitted to trap on other types (e.g. illegal instruction causing SIGILL)
- NOPs provide places to insert new instructions
- Stack maps allow mapping from register / stack values to IR values
- New code paths can be created on demand

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)

Questions?