Compiling Smalltalk to fast native Code

David Chisnall

February 5, 2012

Objective-C: C in Smalltalk Objects

- Created by Brad Cox and Tom Love in 1986 to package C libraries in Smalltalk-like classes
- Smalltalk object model
- C code in methods, message passing between objects
- Rich set of standard class libraries

The Compiler and the Runtime

- The compiler generates calls to functions in the runtime library
- All Smalltalk-like features are implemented by runtime calls
- Calls to C libraries have the same cost as calling them from C
- Can incrementally deploy Objective-C code with C/C++ libraries
- The same model can be used for Smalltalk

Pragmatic Smalltalk

- Dialect of Smalltalk used by Étoilé
- Implements Smalltalk-80 language, but not the standard libraries
- Compiles to native code (JIT or static compiler)
- Emits code ABI-compatible with Objective-C

Compiler Architecture

Clang (Objective-C)

LLVM Optimiser

Executable

LLVM Linker / Optimiser

Native Linker

EScript Parser

LLVM Optimiser

JIT

LanguageKit

Interpreter

Smalltalk Parser

Execution Architecture

Smalltalk Applications	ObjC Applications
SmalltalkSupport / LanguageKitRuntime / ObjC Frameworks	
	GNUstep AppKit
GNUstep Foundation	X11
libobjc	C/C++ Libraries
libc	
Kernel	

LanguageKit

- AST for representing Smalltalk-like languages
- Interpreter for directly executing the AST
- LLVM-based code generation back end for compiling
- Written in Objective-C

Statically Compiled Smalltalk

- Each method is compiled to a function with receiver and selector as first two arguments
- Each class is a structure with a specific layout containing introspection information

Compiling Smalltalk: The Hard Bits

- Small integers
- Blocks
- Non-local returns
- Memory management

Small Objects

- Objects hidden in pointers (e.g. small integers)
- Very common operations implemented as (hand optimised) C functions
- Inlined during compilation
- Very fast: almost the same speed as C integer arithmetic - Fibonacci benchmark ran the same speed as GCC 4.2.1

Blocks

- Objective-C now supports blocks
- LanguageKit uses the same ABI
- Smalltalk and Objective-C blocks can be used interchangeably.

Non-Local Returns

- Returns from blocks
- Ugly feature, should never have been allowed in the language
- Implemented using same mechanism as exceptions (DWARF stack unwinding)
- Very slow, but offset by inlining common operations (e.g. ifTrue:) so that they don't actually use this mechanism
- If you're not mixing with C code, could be faster...

Memory Management

- Objective-C can use tracing GC or reference counting
- LanguageKit supports GC or automatic reference counting (ARC)
- Optimisation passes remove redundant retain / release operations in ARC mode.

Sending Messages to C

Writing a method just to call a C function is cumbersome (and slow!)

```
"Smalltalk code:"
C sqrt: 42.
C fdim: {60. 12}.
C NSLocation: l InRange: r.
```

Generates exactly the same code as:

```
// C code
sqrt(42);
fdim(60, 12);
NSLocationInRange(l, r);
```

No bridging code, no custom wrappers, just native function calls in the compiled code.

What Makes Things Slow?

- Small integer arithmetic
- Boxing
- Dynamic message lookup
- Memory management operations

Small Objects Support in Libobjc

- Allows Smalltalk SmallInts to be returned to ObjC code (no boxing required)
- Removes the need for LanguageKit to add conditionals around every message send
- Approximately 40% reduction in code size
- Smaller code means better instruction cache usage

Cost of Small Ints

- Around 5-10% on the cost of a message send
- Offset by smaller code, better cache usage
- Offset by fewer memory allocations / deallocations
- On 64-bit platforms we also have two flavours of double and 7-character ASCII strings inside pointers.

Lookup Caching

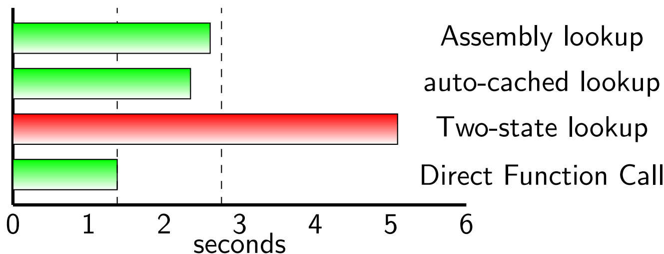
- New lookup function returns a pointer to a structure
- Structure contains a version
- Version incremented if the method is replaced
- Safe automatic caching now possible
- Optimisation pass caches all lookups in loops and to classes

Optimising Lookup More

- Traditional lookup is two phases - return the function, call the function
- Better version would talk-call the method
- Not possible to implement in C, requires some assembly code.

Message Sending

Message sending loop on an 800MHz ARM Cortex A8



Approximately 23,000,000 messages per second on a slow machine.

Speculative Inlining

- C can insert copies of functions where they are used
- Smalltalk message sends may map to different methods
- But we can guess one...
- ...inline it...
- ...and wrap it in a test to see if we guessed right
- Lots of other optimisations (e.g. constant propagation) benefit from inlining!

Type Inference / Feedback

- Moving between integer and float registers is expensive (pipeline stall on ARM)
- If you're doing a lot of floating-point arithmetic, then determining the

Smalltalk Requires Garbage Collection

- General solutions are always slower than specific ones in microbenchmarks
- Usually this is amortised because no one writes a good implementation everywhere
- How do we do GC fast?

Objective-C Memory Management

Traditionally uses manual reference counting + autorelease pools, now automatically inserted by compiler.

Advantages:

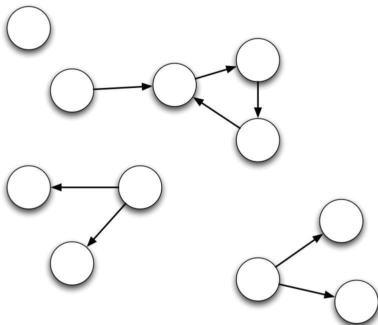
- Relatively low overhead.
- Deterministic overhead.
- No overhead for stack assignments due to autorelease pools

Disadvantages:

- Cycles leak unless you manually break them.
- Relatively expensive to do heap (e.g. ivar) assignments

Garbage Collection 101

The problem: Some of these objects are no longer needed. Which ones?

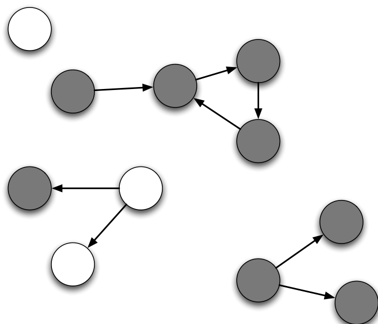


Approach 1: Tracing

- Mark some locations as roots (stack, globals)

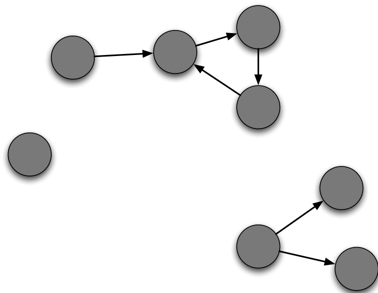
Approach 1: Tracing

- Mark some locations as roots (stack, globals)
- Follow all pointers, colouring objects as you go.



Approach 1: Tracing

- Mark some locations as roots (stack, globals)
- Follow all pointers, colouring objects as you go.
- Delete everything you didn't visit.



Approach 1: Tracing

Advantages

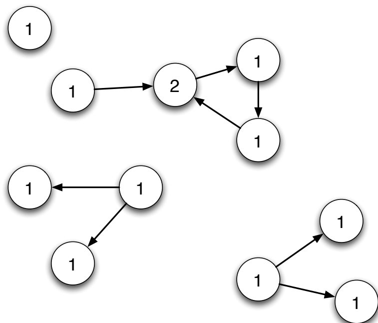
- No problems with cycles.
- Needs to track every object location (one pointer per object) to free the unvisited ones.

Disadvantages:

- Tracing the whole of memory can take a long time, longer the more memory you use. Generations offset this.
- Doesn't play well with swap. All RAM is working set, must be swapped in for a full sweep.
- Doesn't play well with CPU cache. Data cache fills up with objects that are not accessed by anything other than the GC.
- Nondeterministic performance.

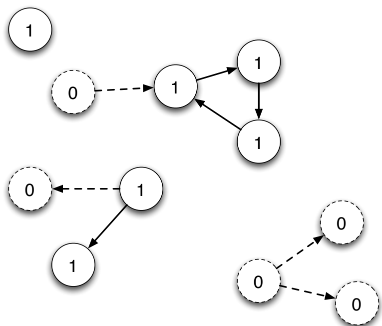
Approach 2: Reference Counting

- Keep a count of all pointers to an object.



Approach 2: Reference Counting

- Keep a count of all pointers to an object.
- Delete objects when their reference count hits 0.



Approach 2: Reference Counting

- Keep a count of all pointers to an object.
- Delete objects when their reference count hits 0.
- Leak cycles.

Approach 2a: Reference Counting and Cycle Detection

If an object is released, but not freed, follow all pointers from it and see if you find enough pointers back to account for its reference count, delete it if you do.

Advantages:

- Deterministic performance.
- Good locality of reference.
- Scales well to NUMA systems.

Disadvantages:

- Overhead on every (heap) assignment.