

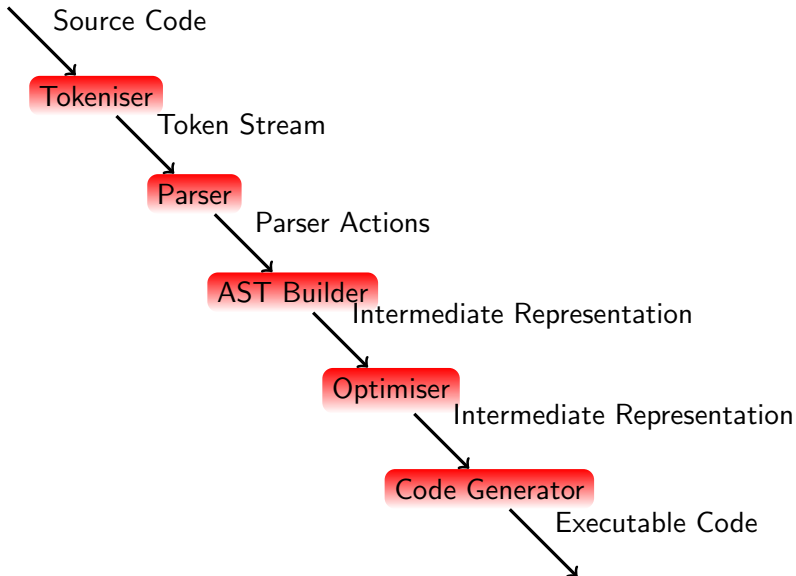
Modern Intermediate Representations (IR)

David Chisnall
University of Cambridge
LLVM Summer School, Paris, June 12, 2017

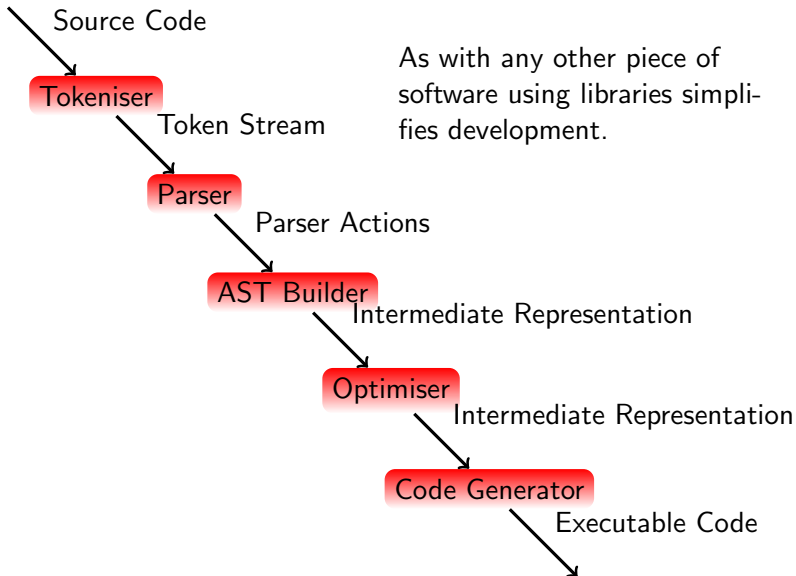
Reusable IR

- Modern compilers are made from loosely coupled components
- Front ends produce IR
- Middle 'ends' transform IR (optimisation / analysis / instrumentation)
- Back ends generate native code (object code or assembly)

Structure of a Modern Compiler



Structure of a Modern Compiler



Optimisation Passes

- Modular, transform IR (Analysis passes just inspect IR)
- Can be run multiple times, in different orders
- May not always produce improvements when run in the wrong order!
- Some intentionally pessimise code to make later passes work better

Register vs Stack IR

- Stack makes interpreting, naive compilation easier
- Register makes various optimisations easier
- Which ones?

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r4 + r5  
r7 = r3 * r6  
store a r6
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r5  
r7 = r3 * r6  
store a r7
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r2  
r7 = r3 * r6  
store a r7
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b  
r2 = load c  
r3 = r1 + r2  
r4 = load b  
r5 = load c  
r6 = r1 + r2  
r7 = r3 * r3  
store a r7
```

Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b  
load c  
add  
load b  
load c  
add  
mul  
store a
```

Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b  
load c  
add  
dup  
mul  
store a
```


Problems with CSE and Stack IR

- Entire operation must happen at once (no incremental algorithm)
- Finding identical subtrees is possible, reusing results is harder
- If the operations were not adjacent, must spill to temporary

Hierarchical vs Flat IR

- Source code is hierarchical (contains structured flow control, scoped values)
- Assembly is flat (all flow control is by jumps)
- Intermediate representations are supposed to be somewhere between the two
- Think about the possible ways that a `for` loop, `while` loop, and `if` statement with a backwards `goto` might be represented.

Hierarchical IR

- Easy to express high-level constructs
- Preserves program semantics
- Preserves high-level semantics (variable lifetime, exceptions) clearly
- Example: WHRIL in MIPSPro/Open64/Path64 and derivatives

Flat IR

- Easy to map to the back end
- Simple for optimisations to process
- Must carry scope information in ad-hoc ways (e.g. LLVM IR has intrinsics to explicitly manage lifetimes for stack allocations)
- Examples: LLVM IR, CGIR, PTX

Questions?

LLVM IR and Transform Pipeline

David Chisnall
University of Cambridge
LLVM Summer School, Paris, June 12, 2017

What Is LLVM IR?

- Unlimited Single-Assignment Register machine instruction set
- Strongly typed
- Three common representations:
 - Human-readable LLVM assembly (.ll files)
 - Dense 'bitcode' binary representation (.bc files)
 - C++ classes

Unlimited Register Machine?

- Real CPUs have a fixed number of registers
- LLVM IR has an infinite number
- New registers are created to hold the result of every instruction
- CodeGen's register allocator determines the mapping from LLVM registers to physical registers
- Type legalisation maps LLVM types to machine types and so on (e.g. 128-element float vector to 32 SSE vectors or 16 AVX vectors, 1-bit integers to 32-bit values)

Static Single Assignment

- Registers may be assigned to only once
- Most (imperative) languages allow variables to be... variable
- This requires some effort to support in LLVM IR: SSA registers are not variables
- SSA form makes dataflow explicit: All consumers of the result of an instruction read the output register(s)

Multiple Assignment

```
int a = someFunction();  
a++;
```

- One variable, assigned to twice.

Translating to LLVM IR

```
%a = call i32 @someFunction()  
%a = add i32 %a, 1
```

error: multiple definition of local value named 'a'

```
  %a = add i32 %a, 1  
    ^
```

Translating to *Correct* LLVM IR

```
%a = call i32 @someFunction()  
%a2 = add i32 %a, 1
```

- Front end must keep track of which register holds the current value of a at any point in the code
- How do we track the new values?

Translating to LLVM IR The Easy Way

```
; int a
;a = alloca i32, align 4
; a = someFunction
%0 = call i32 @someFunction()
store i32 %0, i32* %a
; a++
%1 = load i32* %a
%2 = add i32 %1, 1
store i32 %2, i32* %a
```

- Numbered register are allocated automatically
- Each expression in the source is translated without worrying about data flow
- Memory is not SSA in LLVM

Isn't That Slow?

- Lots of redundant memory operations
- Stores followed immediately by loads
- The Scalar Replacement of Aggregates (SROA) or mem2reg pass cleans it up for us

```
%0 = call i32 @someFunction()  
%1 = add i32 %0, 1
```

Important: SROA only works if the `alloca` is declared in the entry block to the function!

Sequences of Instructions

- A sequence of instructions that execute in order is a *basic block*
- Basic blocks must end with a terminator
- Terminators are *intraprocedural* flow control instructions.
- `call` is not a terminator because execution resumes at the same place after the call
- `invoke` is a terminator because flow either continues or branches to an exception cleanup handler
- This means that even “zero-cost” exceptions can have a cost: they complicate the control-flow graph (CFG) within a function and make optimisation harder.

Intraprocedural Flow Control

- Assembly languages typically manage flow control via jumps / branches (often the same instructions for inter- and intraprocedural flow)
- LLVM IR has conditional and unconditional branches
- Branch instructions are terminators (they go at the end of a basic block)
- Basic blocks are branch targets
- You can't jump into the middle of a basic block (by the definition of a basic block)

What About Conditionals?

```
int b = 12;  
if (a)  
    b++;  
return b;
```

- Flow control requires one basic block for each path
- Conditional branches determine which path is taken

'Phi, my lord, phi!' - Lady Macbeth, Compiler Developer

- ϕ nodes are special instructions used in SSA construction
- Their value is determined by the preceding basic block
- ϕ nodes must come before any non- ϕ instructions in a basic block
- In code generation, ϕ nodes become a requirement for one basic block to leave a value in a specific register.
- Alternate representation: named parameters to basic blocks (used in Swift IR)

Easy Translation into LLVM IR

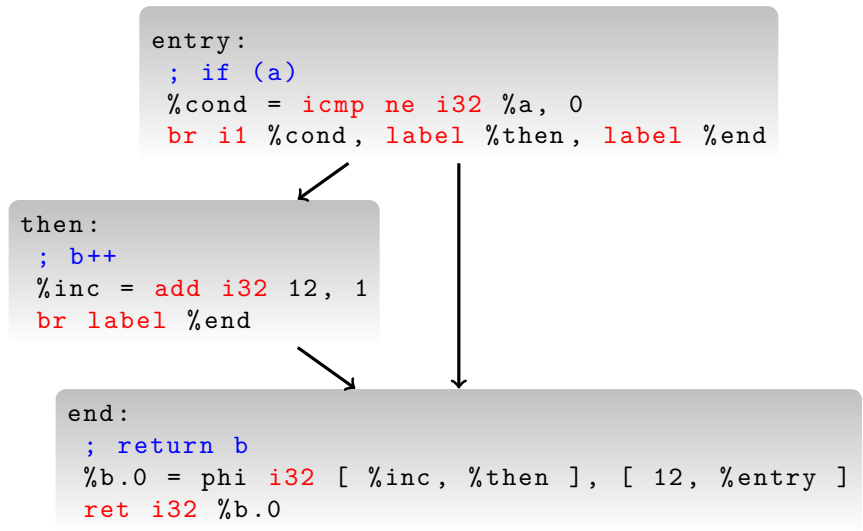
```
entry:  
  ; int b = 12  
  %b = alloca i32  
  store i32 12, i32* %b  
  ; if (a)  
  %0 = load i32* %a  
  %cond = icmp ne i32 %0, 0  
  br i1 %cond, label %then, label %end
```

```
then:  
  ; b++  
  %1 = load i32* %b  
  %2 = add i32 %1, 1  
  store i32 %2, i32* %b  
  br label %end
```

```
end:  
  ; return b  
  %3 = load i32* %b  
  ret i32 %3
```

In SSA Form...

```
entry:  
; if (a)  
%cond = icmp ne i32 %a, 0  
br i1 %cond, label %then, label %end
```



```
then:  
; b++  
%inc = add i32 12, 1  
br label %end
```

```
end:  
; return b  
%b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]  
ret i32 %b.0
```

In SSA Form...

```
entry:  
; if (a)  
%cond = icmp ne i32 %a, 0  
br i1 %cond, label %then, label %end
```

```
then:  
; b++  
%inc = add i32 12, 1  
br label %end
```

```
end:  
; return b  
%b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]  
ret i32 %b.0
```

The output from
the mem2reg pass

And After Constant Propagation...

```
entry:  
; if (a)  
%cond = icmp ne i32 %a, 0  
br i1 %cond, label %then, label %end
```

```
then:  
br label %end
```

The output from the
constprop pass. No add
instruction.

```
end:  
; b++  
; return b  
%b.0 = phi i32 [ 13, %then ], [ 12, %entry ]  
ret i32 %b.0
```

And After CFG Simplification...

```
entry:  
  %tobool = icmp ne i32 %a, 0  
  %0 = select i1 %tobool, i32 13, i32 12  
  ret i32 %0
```

- Output from the simplifycfg pass
- No flow control in the IR, just a select instruction

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.
Would a predicated add instruction be better on ARM?

Canonical Form

- LLVM IR has a notion of canonical form
- High-level have a single canonical representation
- For example, loops:
 - Have a single entry block
 - Have a single back branch to the start of the entry block
 - Have induction variables in a specific form
- Some passes generate canonical form from non-canonical versions commonly generated by front ends
- All other passes can expect canonical form as input

Functions

- LLVM functions contain at least one basic block
- Arguments are registers and are explicitly typed
- Registers are valid only within a function scope

```
@hello = private constant [13 x i8] c"Hello
world!\00"

define i32 @main(i32 %argc, i8** %argv) {
entry:
    %0 = getelementptr [13 x i8]* @hello, i32 0,
        i32 0
    call i32 @puts(i8* %0)
    ret i32 0
}
```

Get Element Pointer?

- Often shortened to GEP (in code as well as documentation)
- Represents pointer arithmetic
- Translated to complex addressing modes for the CPU
- Also useful for alias analysis: result of a GEP is the same object as the original pointer (or undefined)

In modern LLVM IR, on the way to typeless pointers, GEP instructions carry the pointee type. For brevity, we'll use the old form in the slides.

F!@£ing GEPs! HOW DO THEY WORK?!?

```
struct a {  
    int c;  
    int b[128];  
} a;  
int get(int i) { return a.b[i]; }
```

Flattening GEPs! HOW DO THEY WORK?!?

```
struct a {  
    int c;  
    int b[128];  
} a;  
int get(int i) { return a.b[i]; }
```

```
%struct.a = type { i32, [128 x i32] }  
@a = common global %struct.a zeroinitializer,  
    align 4  
  
define i32 @get(i32 %i) {  
entry:  
    %arrayidx = getelementptr inbounds %struct.a*  
        @a, i32 0, i32 1, i32 %i  
    %0 = load i32* %arrayidx  
    ret i32 %0  
}
```

As x86 Assembly

```
define i32 @get(i32 %i) {  
entry:  
    %arrayidx = getelementptr inbounds %struct.a*  
        @a, i32 0, i32 1, i32 %i  
    %0 = load i32* %arrayidx  
    ret i32 %0  
}
```

```
get:  
    movl    4(%esp), %eax        # load parameter  
    movl    a+4(,%eax,4), %eax   # GEP + load  
    ret
```

As ARM Assembly

```
define i32 @get(i32 %i) {  
entry:  
    %arrayidx = getelementptr inbounds %struct.a*  
        @a, i32 0, i32 1, i32 %i  
    %0 = load i32* %arrayidx  
    ret i32 %0  
}
```

```
get:  
    ldr    r1, .LCPI0_0        // Load global address  
    add   r0, r1, r0, lsl #2 // GEP  
    ldr   r0, [r0, #4]        // load return value  
    bx   lr  
.LCPI0_0:  
    .long    a
```


The Most Important LLVM Classes

- `Module` - A compilation unit.

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions
- `PassManagerBuilder` - Constructs optimisation pass sequences to run

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions
- `PassManagerBuilder` - Constructs optimisation pass sequences to run
- `ExecutionEngine` - Interface to the JIT compiler

Writing a New Pass

LLVM optimisations are self-contained classes:

- `ModulePass` subclasses modify a whole module
- `FunctionPass` subclasses modify a function
- `LoopPass` subclasses modify a function
- Lots of analysis passes create information your passes can use!

Example Language-specific Passes

ARC Optimisations:

- Part of LLVM
- Elide reference counting operations in Objective-C code when not required
- Makes heavy use of LLVM's flow control analysis

GNUstep Objective-C runtime optimisations:

- Distributed with the runtime.
- Can be used by clang (Objective-C) or LanguageKit (Smalltalk)
- Cache method lookups, turn dynamic into static behaviour if safe

Writing A Simple Pass

- Memoise an expensive library call
- Call maps a string to an integer (e.g. string intern function)
- Mapping can be expensive.
- Always returns the same result.

```
x = example("some_string");
```

```
static int ._cache;  
if (!._cache)  
    ._cache = example("some_string");  
x = ._cache;
```

Declaring the Pass

```
struct MemoiseExample : ModulePass, InstVisitor<
    SimplePass>
{
    ... // Boilerplate, see SimplePass
    /// The function that we're going to memoise
    Function *exampleFn;
    /// The return type of the function
    Type *retTy;
    /// Call sites and their constant string
        arguments
    using ExampleCall = std::pair<CallInst&,std::
        string>;
    /// All of the call sites that we've found
    SmallVector<ExampleCall, 16> sites;
```

The Entry Point

```
/// Pass entry point
bool runOnModule(Module &Mod) override {
    sites.clear();
    // Find the example function
    exampleFn = Mod.getFunction("example");
    // If it isn't referenced, exit early
    if (!exampleFn)
        return false;
    // We'll use the return type later for the
        caches
    retTy = exampleFn->getFunctionType()->
        getReturnType();
    // Find all call sites
    visit(Mod);
    // Insert the caches
    return insertCaches(Mod);
}
```

Finding the Call

```
void visitCallInst(CallInst &CI) {
    if (CI.getCalledValue() == exampleFn)
        if (auto *arg = dyn_cast<GlobalVariable>(
            CI.getOperand(0)->stripPointerCasts()))
            if (auto *init = dyn_cast<
                ConstantDataSequential>(
                    arg->getInitializer()))
                if (init->isString())
                    sites.push_back({CI,
                                    init->getAsString()});
}
```

Creating the Cache

- Once we've found all of the replacement points, we can insert the caches.
- Don't do this during the search - iteration doesn't like the collection being mutated...

```
StringMap<GlobalVariable*> statics;  
for (auto &s : sites) {  
    auto *lookup = &s.first;  
    auto arg = s.second;  
    GlobalVariable *cache = statics[arg];  
    if (!cache) {  
        cache = new GlobalVariable(M, retTy, false,  
            GlobalVariable::PrivateLinkage,  
            Constant::getNullValue(retTy),  
            "_cache");  
        statics[arg] = cache;  
    }  
}
```


Restructuring the CFG

```
auto *preLookupBB = lookup->getParent();
auto *lookupBB =
    preLookupBB->splitBasicBlock(lookup);
BasicBlock::iterator iter(lookup);
auto *afterLookupBB =
    lookupBB->splitBasicBlock(++iter);
preLookupBB->getTerminator()->eraseFromParent();
lookupBB->getTerminator()->eraseFromParent();
auto *phi = PHINode::Create(retTy, 2, "cache",
    &*afterLookupBB->begin());
lookup->replaceAllUsesWith(phi);
```

Adding the Test

```
IRBuilder <> B(beforeLookupBB);
llvm::Value *cachedClass =
    B.CreateBitCast(B.CreateLoad(cache), retTy);
llvm::Value *needsLookup =
    B.CreateIsNull(cachedClass);
B.CreateCondBr(needsLookup , lookupBB ,
    afterLookupBB);
B.SetInsertPoint(lookupBB);
B.CreateStore(lookup , cache);
B.CreateBr(afterLookupBB);
phi->addIncoming(cachedClass , beforeLookupBB);
phi->addIncoming(lookup , lookupBB);
```

A Simple Test

```
int example(char *foo) {
    printf("example(%s)\n", foo);
    int i=0;
    while (*foo)
        i += *(foo++);
    return i;
}
int main(void) {
    int a = example("a_contrived_example");
    a += example("a_contrived_example");
    a += example("a_contrived_example");
    a += example("a_contrived_example");
    a += example("a_contrived_example");
    return a;
}
```

Running the Test

```
$ clang example.c -O2 ; ./a.out ; echo $?
```

```
example(a contrived example)
```

```
example(a contrived example)
```

```
example(a contrived example)
```

```
example(a contrived example)
```

```
example(a contrived example)
```

```
199
```

```
$ clang -Xclang -load -Xclang ./memo.so -O2
```

```
$ ./a.out ; echo $?
```

```
example(a contrived example)
```

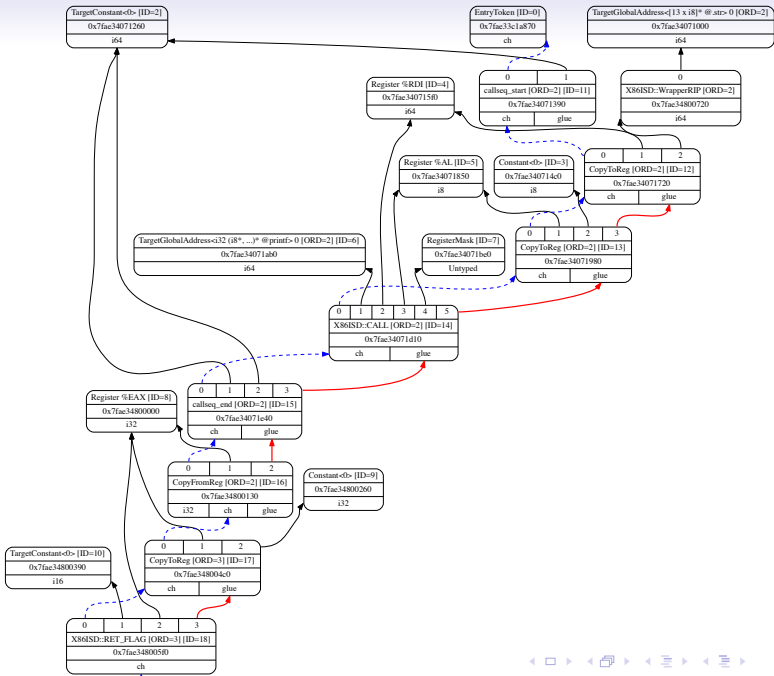
```
199
```

How Does LLVM IR Become Native Code?

- Transformed to directed acyclic graph representation (SelectionDAG)
- Mapped to instructions (Machine IR)
- Streamed to assembly or object code writer

Selection DAG

- DAG defining operations and dependencies
- Legalisation phase lowers IR types to target types
 - Arbitrary-sized vectors to fixed-size
 - Float to integer and softfloat library calls
 - And so on
- DAG-to-DAG transforms simplify structure
- Code is still (more or less) architecture independent at this point
- Some peephole optimisations happen here



Instruction Selection

- Pattern matching engine maps subtrees to instructions and pseudo-ops
- Generates another SSA form: Machine IR (MIR)
- Real machine instructions
- Some (target-specific) pseudo instructions
- Mix of virtual and physical registers
- Low-level optimisations can happen here

Register allocation

- Maps virtual registers to physical registers
- Adds stack spills / reloads as required
- Can reorder instructions, with some constraints

MC Streamer

- Class with assembler-like interface
- Emits one of:
 - Textual assembly
 - Object code file (ELF, Mach-O, COFF)
 - In-memory instruction stream
- All generated from the same instruction definitions

Questions?

Modern Processor Architectures (A compiler writer's perspective)

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

The 1960s - 1970s

- Instructions took multiple cycles
- Only one instruction in flight at once
- Optimisation meant minimising the number of instructions executed
- Sometimes replacing expensive general-purpose instructions with specialised sequences of cheaper ones

The 1980s

- CPUs became pipelined
- Optimisation meant minimising pipeline stalls
- Dependency ordering such that results were not needed in the next instruction
- Computed branches became very expensive when not correctly predicted

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```


Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

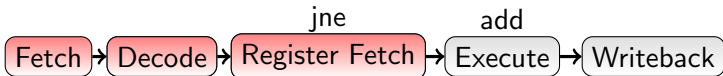
```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Stall Example



```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
...
```

```
add r1, r1, -1
```

```
jne r1, 0, start
```

Fixing the Stall

```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
    add r1, r1, -1
```

```
    ...
```

```
    jne r1, 0, start
```

Fixing the Stall

```
(int i=100 ; i!=0 ; i--)
```

```
...
```

```
start:
```

```
    add r1, r1, -1
```

```
    ...
```

```
    jne r1, 0, start
```

Is this a good solution?

Note about efficiency

- In-order pipelines give very good performance per Watt at low power
- Probably not going away any time soon (see ARM Cortex A7, A53)
- Compiler optimisations can make a big difference!

The Early 1990s

- CPUs became much faster than memory
- Caches hid some latency
- Optimisation meant maximising locality of reference, prefetching
- Sometimes, recalculating results is faster than fetching from memory
- Note: Large caches consume a lot of power, but fetching a value from memory can cost the same as several hundred ALU ops

The Mid 1990s

- CPUs became superscalar
 - Independent instructions executed in parallel
- CPUs became out-of-order
 - Reordered instructions to reduce dependencies
- Optimisation meant structuring code for highest-possible ILP
- Loop unrolling no longer such a big win

Superscalar CPU Pipeline Example: Sandy Bridge

Can dispatch up to six instructions at once, via 6 pipelines:

1. ALU, VecMul, Shuffle, FpDiv, FpMul, Blend
2. ALU, VecAdd, Shuffle, FpAdd
3. Load / Store address
4. Load / Store address
5. Load / Store data
6. ALU, Branch, Shuffle, VecLogic, Blend

Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?

Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?
Only 35%!

Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?

Only 35%!

Only 12% with a 90% hit rate!

The Late 1990s

- Single Instruction, Multiple Data (SIMD) became mainstream
- e.g. $1,2,3,4 + 2,3,4,5 = 3,5,7,9$ as a single instruction
- SIMD units are also superscalar, so potentially multiple SIMD instructions dispatched per cycle!
- Factor of 2-4 \times speedup when used correctly
- Optimisation meant ensuring data parallelism
- Loop unrolling starts winning again, as it exposes later optimisation opportunities
- Modern compilers unroll near the start and then re-roll loops near the end of the optimisation pipeline!

The Early 2000s

- (Homogeneous) Multicore became mainstream
- Power efficiency became important
- Parallelism provides both better throughput and lower power
- Optimisation meant exploiting coarse-grained parallelism

False sharing and multicore

- Thread a writes to one variable.
- Thread b reads another.
- Both variables are in the same cache line.
- What happens?

Cache coherency: MESI protocol

Modified cache line contains local changes

Shared cache line contains shared (read-only) copies of data

Exclusive cache line contains the only copy of data

Invalid cache line contains invalid data

Cache ping-pong

- Both cores have the line in shared state
- Core 2 acquires exclusive state (around 200 cycles), Core 1 line moves to invalid state
- Core 2 writes, line enters modified state
- Core 1 reads, fetches data from core 2 (around 200 cycles), both lines move to shared state

The Late 2000s

- Programmable GPUs became mainstream
- Hardware optimised for stream processing in parallel
- Very fast for massively-parallel floating point operations
- Cost of moving data between CPU and GPU is high
- Optimisation meant offloading operations to the GPU

The 2010s

- Modern processors come with multiple CPU and GPU cores
- All cores behind the same memory interface, cost of moving data between them is low
- Increasingly contain specialised accelerators
- Often contain general-purpose (programmable) cores for specialised workload types (e.g. DSPs)
- Optimisation is hard.
- Lots of jobs for compiler writers!

Common Programming Models

- Sequential (can we automatically detect parallelism)?
- Explicit message passing (e.g. MPI, Erlang)
- Annotation-driven parallelism (e.g. OpenMP)
- Explicit task-based parallelism (e.g. libdispatch)
- Explicit threading (e.g. pthreads, shared-everything concurrency)

Parallelising Loop Iterations

- Same problem to targeting SIMD
- Looser constraints: data can be unaligned, flow control can be independent
- Tighter constraints: loop iterations must be completely independent
- (Usually) more overhead for creating threads than using SIMD lanes

Communication and Synchronisation Costs

What's the best implementation strategy for this?

```
#pragma omp parallel for
for (int i=0 ; i<100 ; i++)
{
    process(a[i]);
}
```


Communication and Synchronisation Costs

What's the best implementation strategy for this?

```
#pragma omp parallel for
for (int i=0 ; i<100 ; i++)
{
    process(a[i]);
}
```

- Spawn a new thread for each iteration?
- Spawn one thread per core, split loop iterations between them?
- Spawn one thread per core, have each one start a loop iteration and check the current loop induction variable before doing the next one?
- Spawn one thread per core, pass batches of loop iterations to each one?
- Something else?

HELIX: Parallelising Sequential Segments in Loops

- Loop iterations each run a sequence of (potentially expensive) steps
- Run each step on a separate core
- Each core runs the same number of iterations as the original loop
- Use explicit synchronisation to detect barriers

Execution Models for GPUs

- GPUs have no standardised public instruction set
- Code shipped as source or some portable IR
- Compiled at install or load time
- Loaded to the device to run

SPIR

- Standard Portable Intermediate Representation
- Khronos Group standard, related to OpenCL
- Subsets of LLVM IR (one for 32-bit, one for 64-bit)
- Backed by ARM, AMD, Intel (everyone except nVidia)
- OpenCL programming model extensions as intrinsics
- Design by committee nightmare, no performance portability

SPIR-V

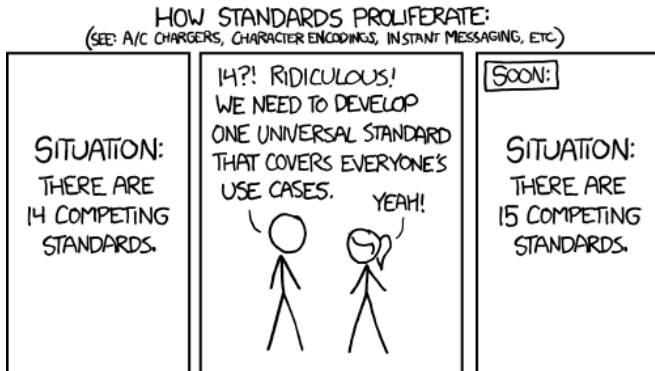
- Standard Portable Intermediate Representation (for Vulkan)
- Khronos Group standard, related to Vulkan
- Independent encoding, easy to map to/from LLVM IR
- Backed by ARM, AMD, Intel *and* nVidia
- Intended as a compilation target for GLSL, OpenCL C, others

PTX

- Parallel Thread eXecution
- IR created by nVidia
- Semantics much closer to nVidia GPUs

HSAIL

- Heterogeneous Systems Architecture Intermediate Language
- Cross-vendor effort under the HSA umbrella
- More general than PTX (e.g. allows function pointers)



Single Instruction Multiple Thread (SIMT)

- SIMD with independent register sets, varying-sized vectors
- Program counter (PC) shared across threads
- All threads perform the same operation, but on different data
- Diverging threads get their own PC
- Only one PC used at a time
- Throughput halves for each divergent branch until only one thread is running
- Explicit barrier instructions allow diverged threads to rendezvous.

Thread Groups

- GPU programs run the same code (kernel) on every element in an input set
- Threads in a group can communicate via barriers and other synchronisation primitives
- Thread groups are independent

GPU Memory Model

- Per-thread memory (explicitly managed, equivalent to CPU cache)
- Shared memory between thread groups (equivalent to CPU shared L3 cache)
- Global memory (read-write, cache coherent)
- Texture memory (read-only or write-only, non-coherent)

Costs for GPU Use

- Setup context (MMU mappings on GPU, command queue). Typically once per application.
- Copying data across the bus is very expensive, may involve bounce buffers
- Newer GPUs share a memory controller with the CPU (might not share an address space, setting IOMMU mappings can be expensive)
- Calling into the OS kernel to send messages (userspace command submission helps here)
- Synchronisation (cache coherency) between CPU and GPU

Thought Experiment: memcpy(), memset()

- GPUs and DSPs are fast stream processors
- Ideal for things like memcpy(), memset()
- What bottlenecks prevent offloading all memset() / memcpy() calls to a coprocessor?
- How could they be fixed?

Autoparallelisation vs Autovectorisation

- Autovectorisation is a special case of autoparallelisation
- Requires dependency, alias analysis between sections
- GPU SIMT processors are suited to the same sorts of workloads as SIMD coprocessors
- (Currently) only sensible when working on large data or very expensive calculations

Loop offloading

- Identify all inputs and outputs
- Copy all inputs to the GPU
- Run the loop as a GPU kernel
- Copy all outputs back to main memory
- Why can this go wrong?

Loop offloading

- Identify all inputs and outputs
- Copy all inputs to the GPU
- Run the loop as a GPU kernel
- Copy all outputs back to main memory
- Why can this go wrong?
- What happens if you have other threads accessing memory?
- Shared everything is hard to reason about

Avoiding Divergent Flow Control: If Conversion

- Two threads taking different paths must be executed sequentially
- Execute both branches
- Conditionally select the result
- Also useful on superscalar architectures - reduces branch predictor pressure
- Early GPUs did this in hardware

OpenCL on the CPU

- Can SIMD emulate SIMT?
- Hardware is similar, SIMT is slightly more flexible
- Sometimes, OpenCL code runs faster on the CPU if data is small
- Non-diverging flow is trivial
- Diverging flow requires special handling

Diverging Flow

- Explicit masking for if conversion
- Each possible path is executed
- Results are conditionally selected
- Significant slowdown for widely diverging code
- Stores, loads-after-stores require special handling

OpenCL Synchronisation Model

- Explicit barriers block until all threads in a thread group have arrived.
- Atomic operations (can implement spinlocks)
 - Why would spinlocks on a GPU be slow?

OpenCL Synchronisation Model

- Explicit barriers block until all threads in a thread group have arrived.
- Atomic operations (can implement spinlocks)
 - Why would spinlocks on a GPU be slow?
 - Branches are slow, non-streaming memory-access is expensive...
 - Random access to workgroup-shared memory is cheaper than texture memory

Barriers and SIMD

- Non-diverging flow, barrier is a no-op
- Diverging flow requires rendezvous
- Pure SIMD implementation (single core), barrier is where start of a basic block after taking both sides of a branch
- No real synchronisation required

OpenCL with SIMD on multicore CPUs

- Barriers require real synchronisation
- Can be a simple pthread barrier
- Alternatively, different cores can run independent thread groups

Questions?

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

```
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

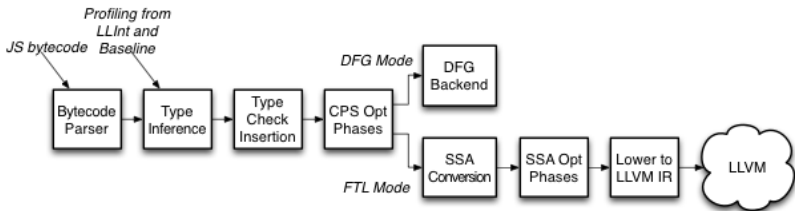
- Pioneered by Self
- Both LLInt and the baseline JIT collect type information
- Later tiers can optimise based on this
- More useful than type inference for optimisation (this is usually type X, vs this must always be type X, Y, or Z)

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). LLVM8 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

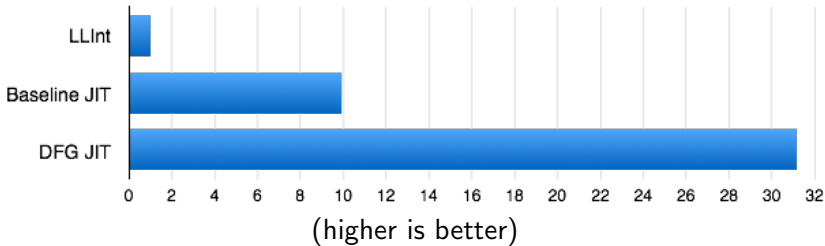
- Static type inference is really hard for dynamic languages
- Must be conservative: bad for optimisation
- Type feedback provided by earlier tiers
- Propagate forwards (e.g. `int32 + int32` is probably `int32`: overflow unlikely)
- Fed back into later compiler stages
- LLInt and baseline JIT collect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



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.

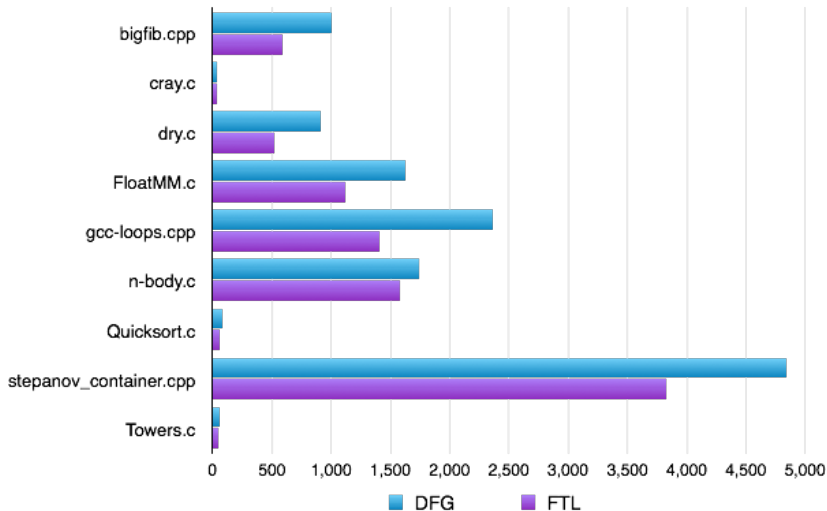
Patchpoints for deoptimisation

- 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

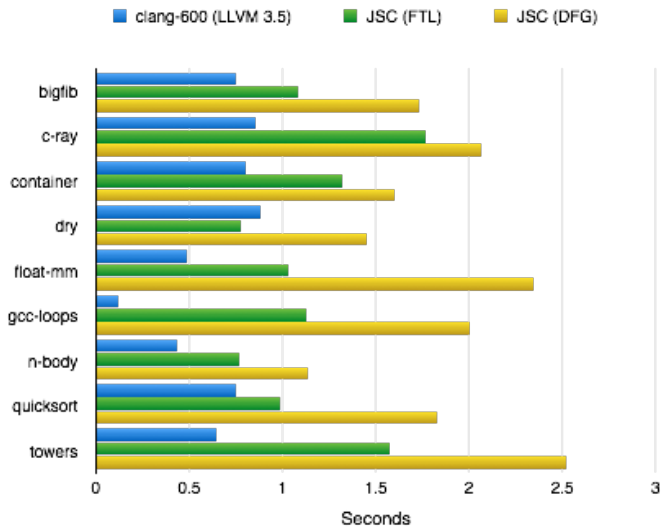
- Speculatively compiled assuming fixed field offsets
- Can become incorrect as more code is executed
- Dynamically patched with correct offsets when hit

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