AN LLVM BACKEND FOR GHC

David Terei & Manuel Chakravarty
University of New South Wales

Haskell Symposium 2010
Baltimore, USA, 30th Sep

What is ‘An LLVM Backend’?

- Low Level Virtual Machine
- Backend Compiler framework
- Designed to be used by compiler developers, not by software developers
- Open Source
- Heavily sponsored by Apple

Motivation

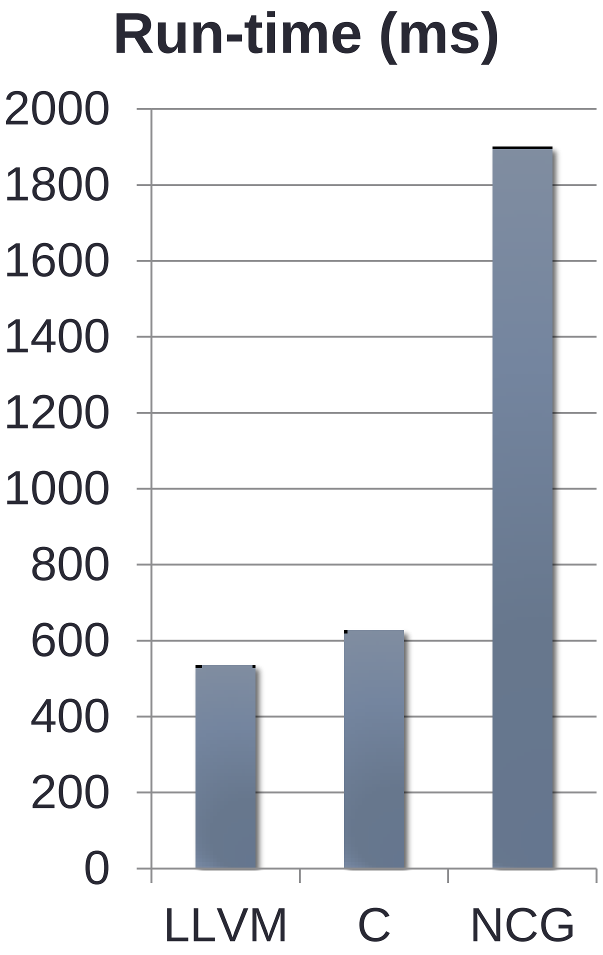
- Simplify
  - Reduce ongoing work
  - Outsource!
- Performance
  - Improve run-time

Example

```
collat :: Int -> Word32 -> Int
collat c 1 = c
collat c n | even n =
             collat (c+1) $ n `div` 2
           | otherwise =
             collat (c+1) $ 3 * n + 1

pmax x n = x `max` (collat 1 n, n)

main = print $ foldl pmax (1,1)
                 [2..1000000]
```



Different, updated run-times compared to paper

Competitors

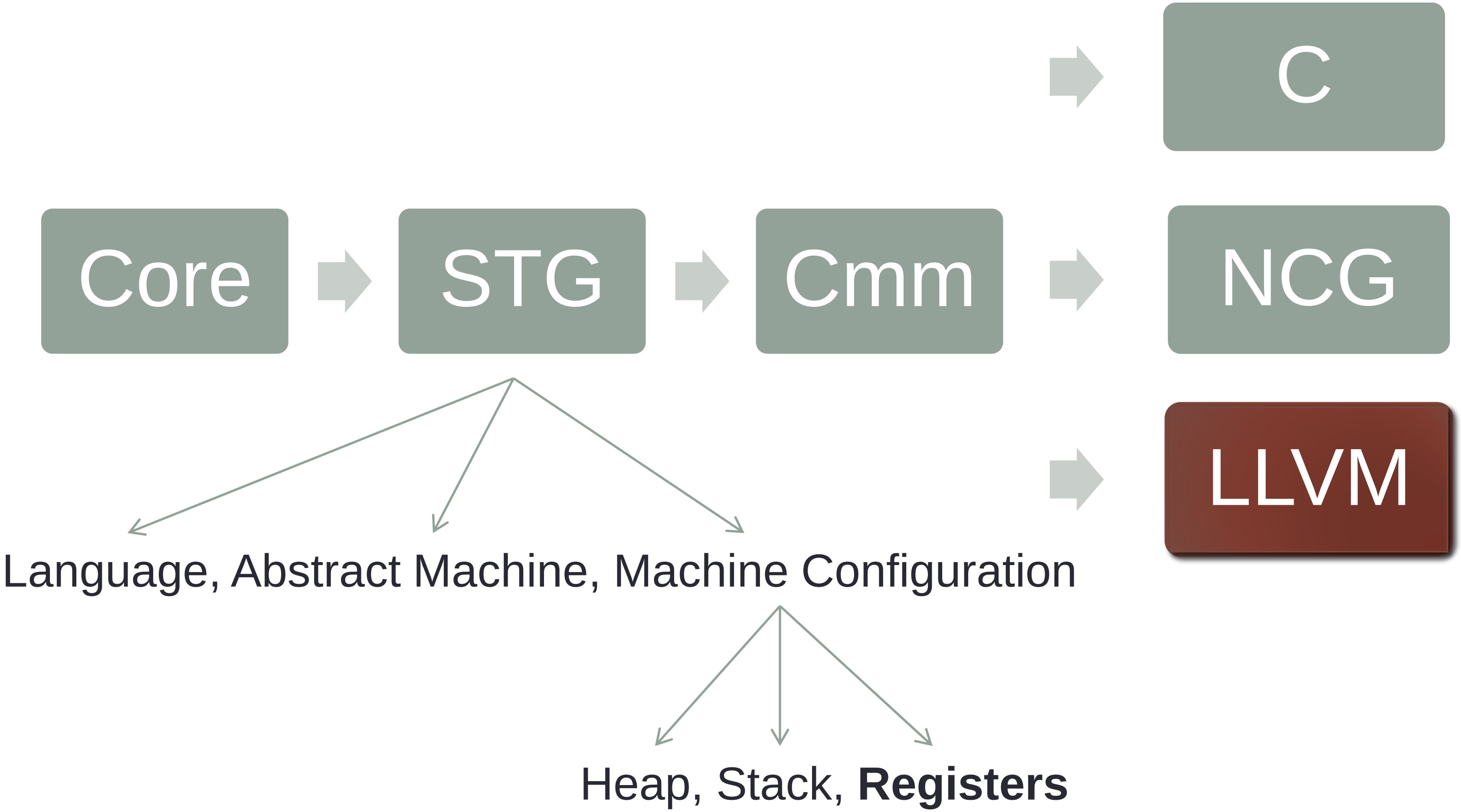
C Backend (C)

- GNU C Dependency
  - Badly supported on platforms such as Windows
- Use a Mangler on assembly code
- Slow compilation speed
  - Takes twice as long as the NCG

Native Code Generator (NCG)

- Huge amount of work
- Very limited portability
- Does very little optimisation work

GHC's Compilation Pipeline



Core

STG

Cmm

C

NCG

LLVM

Language, Abstract Machine, Machine Configuration

Heap, Stack, **Registers**

Compiling to LLVM

Won’t be covering, see ***paper*** for full details:

- Why from Cmm and not from STG/Core
- LLVM, C-- & Cmm languages
- Dealing with LLVM’s SSA form
- LLVM type system

Will be covering:

- Handling the STG Registers
- Handling GHC’s Table-Next-To-Code optimisation

Handling the STG Registers

STG Register	X86 Register
Base	ebx
Heap Pointer	edi
Stack Pointer	ebp
R1	esi

Implement either by:

- In memory
- Pin to hardware registers

NCG?

- Register allocator permanently stores STG registers in hardware

C Backend?

- Uses GNU C extension (*global register variables)* to also permanently store STG registers in hardware

Handling the STG Registers

LLVM handles by implementing a new calling convention:

STG Register	X86 Register
Base	ebx
Heap Pointer	edi
Stack Pointer	ebp
R1	esi

```
define f ghc_cc (Base, Hp, Sp, R1) {
  …
  tail call g ghc_cc (Base, Hp’, Sp’, R1’);
  return void;
}
```

Handling the STG Registers

- **Issue:** If implemented naively then all the STG registers have a live range of the entire function.

- Some of the STG registers can never be scratched (e.g Sp, Hp…) but many can (e.g R2, R3…).

- We need to somehow tell LLVM when we no longer care about an STG register, otherwise it will spill and reload the register across calls to C land for example.

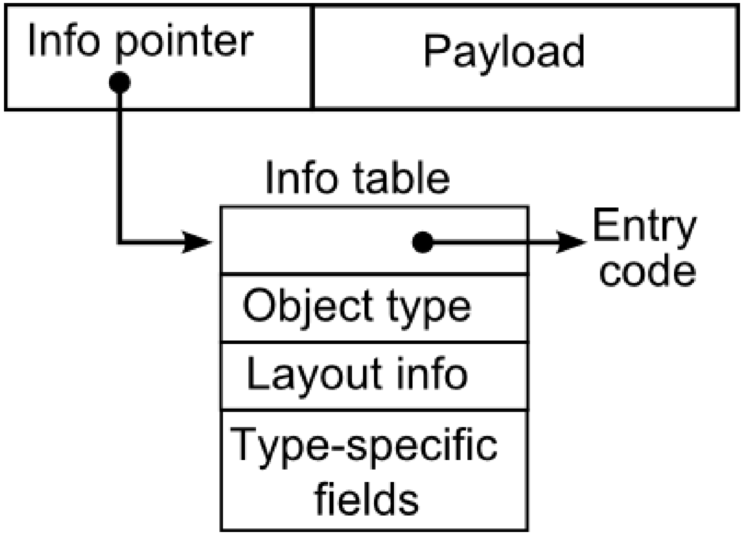
Handling the STG Registers

- We handle this by storing ***undef*** into the STG register when it is no longer needed. We **manually scratch** them.

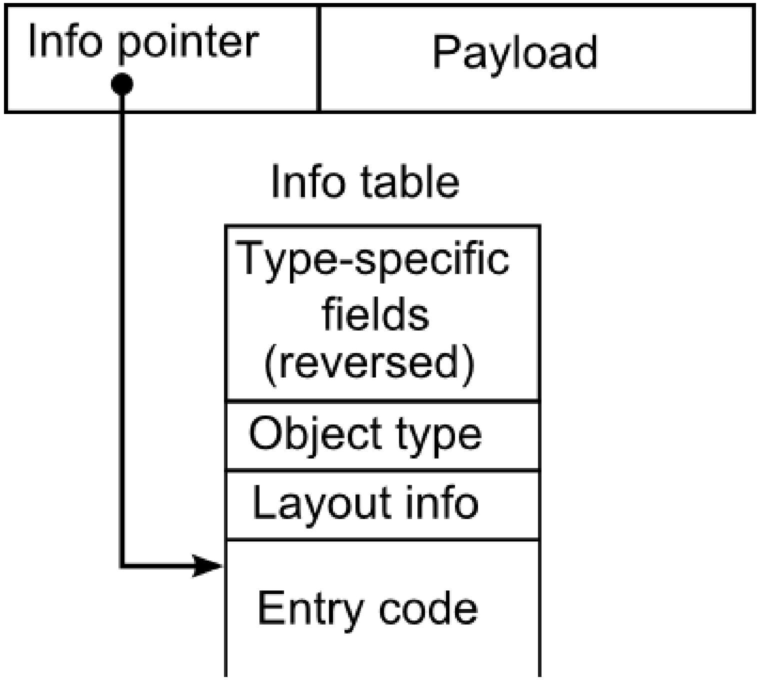
```
define f ghc_cc (Base, Hp, Sp, R1, R2, R3, R4) {
  …
  store undef %R2
  store undef %R3
  store undef %R4
  call c_cc sin(double %f22);
  …
  tail call ghc_cc g(Base,Hp’,Sp’,R1’,R2’,R3’,R4’);
  return void;
}
```

Handling Tables-Next-To-Code

Un-optimised Layout



Optimised Layout



How to implement in LLVM?

Handling Tables-Next-To-Code

Use GNU Assembler ***sub-section*** feature.

- Allows code/data to be put into numbered sub-section
- Sub-sections are appended together in order
- Table in **<*n*>**, entry code in **<*n+1*>**

```
.text 12
sJ8_info:
    movl ...
    movl ...
    jmp  ...

[...]
.text 11
sJ8_info_itable:
    .long ...
    .long 0
    .long 327712
```

Handling Tables-Next-To-Code

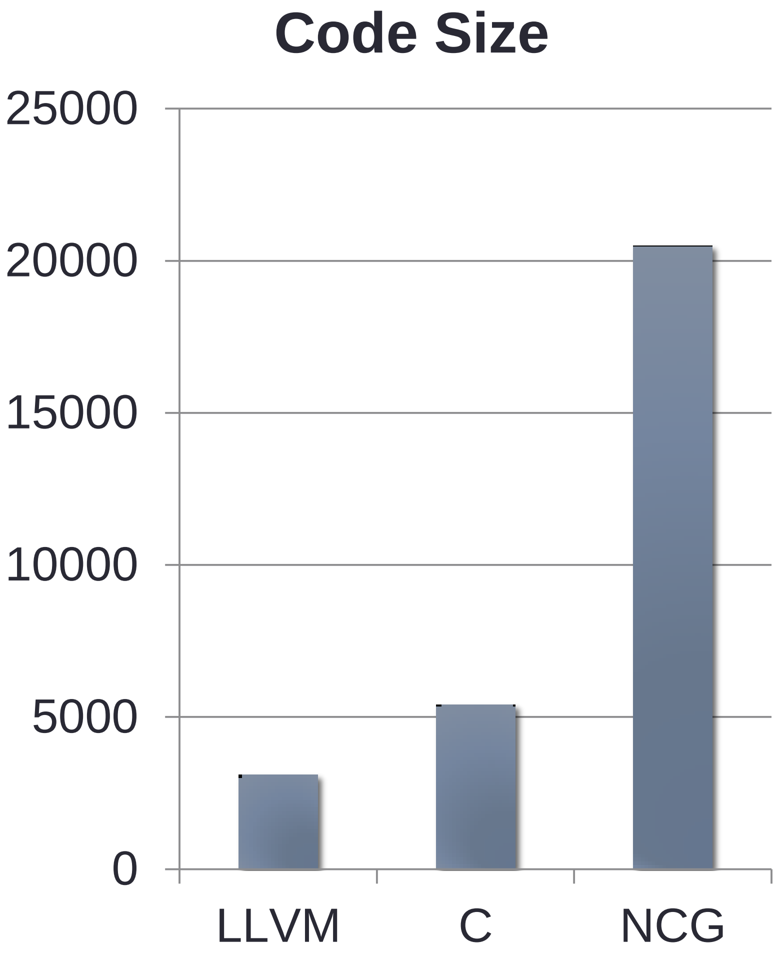
LLVM Mangler

- 180 lines of Haskell (half is documentation)
- Needed only for OS X

C Mangler

- 2,000 lines of Perl
- Needed for every platform

Evaluation: Simplicity



LLVM

- Half of code is representation of LLVM language

C

- Compiler: 1,100 lines
- C Headers: 2,000 lines
- Perl Mangler: 2,000 lines

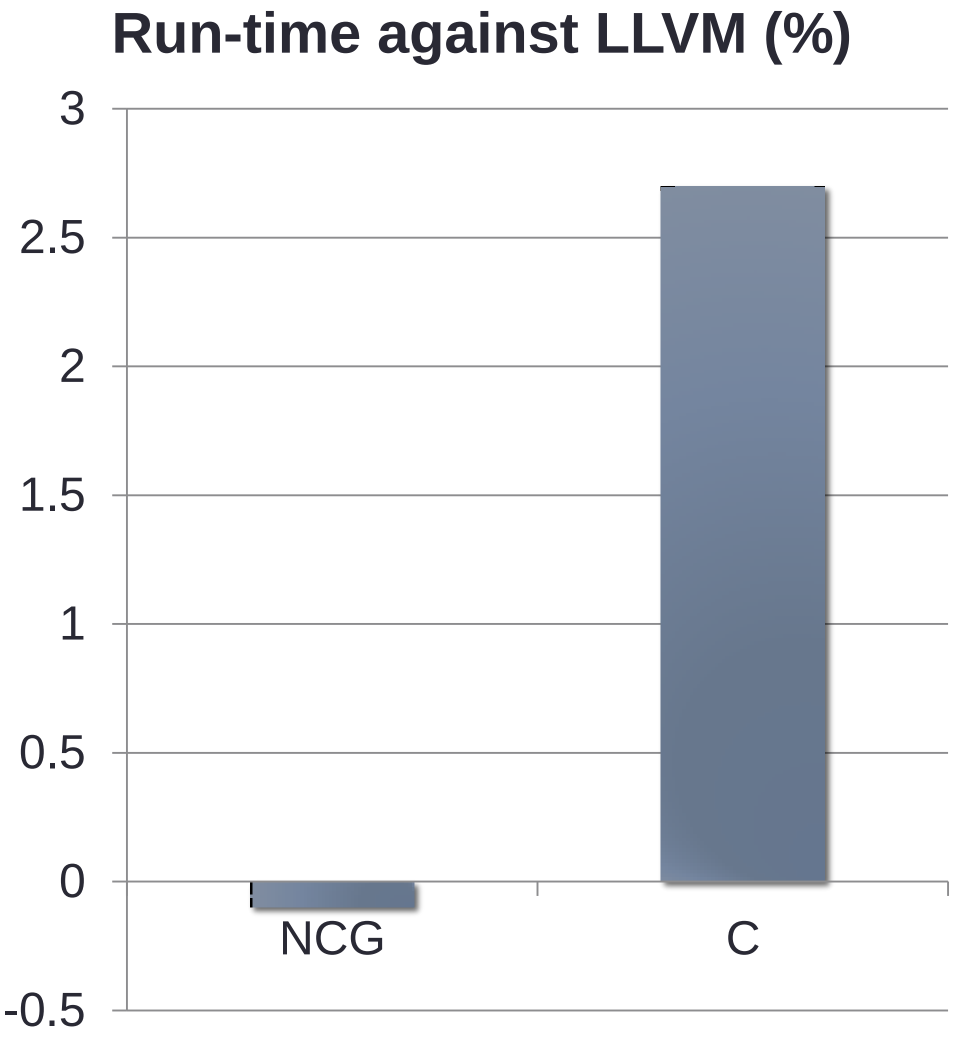
NCG

- Shared component: 8,000 lines
- Platform specific: 4,000 – 5,000 for X86, SPARC, PowerPC

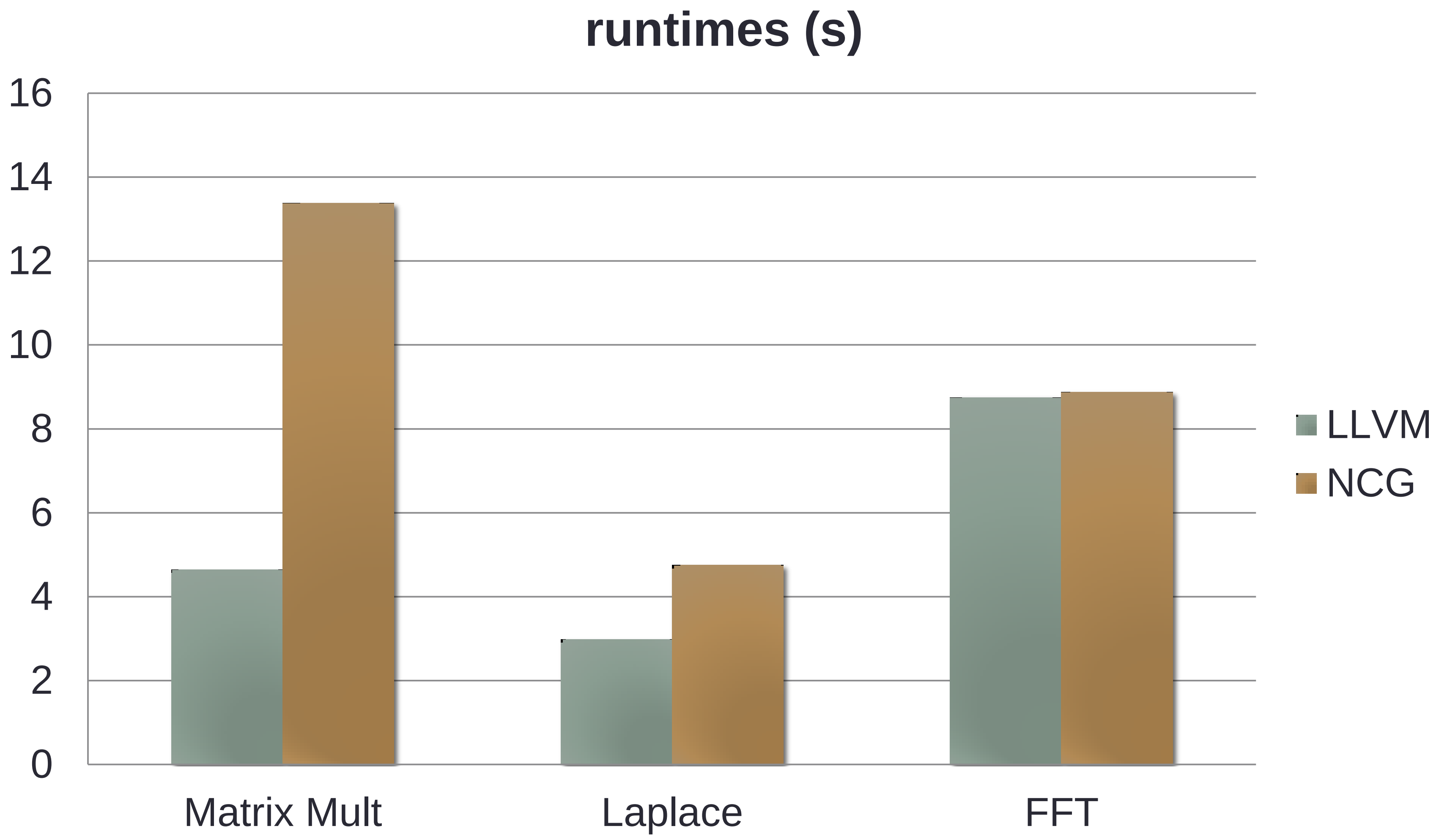
Evaluation: Performance

Nofib:

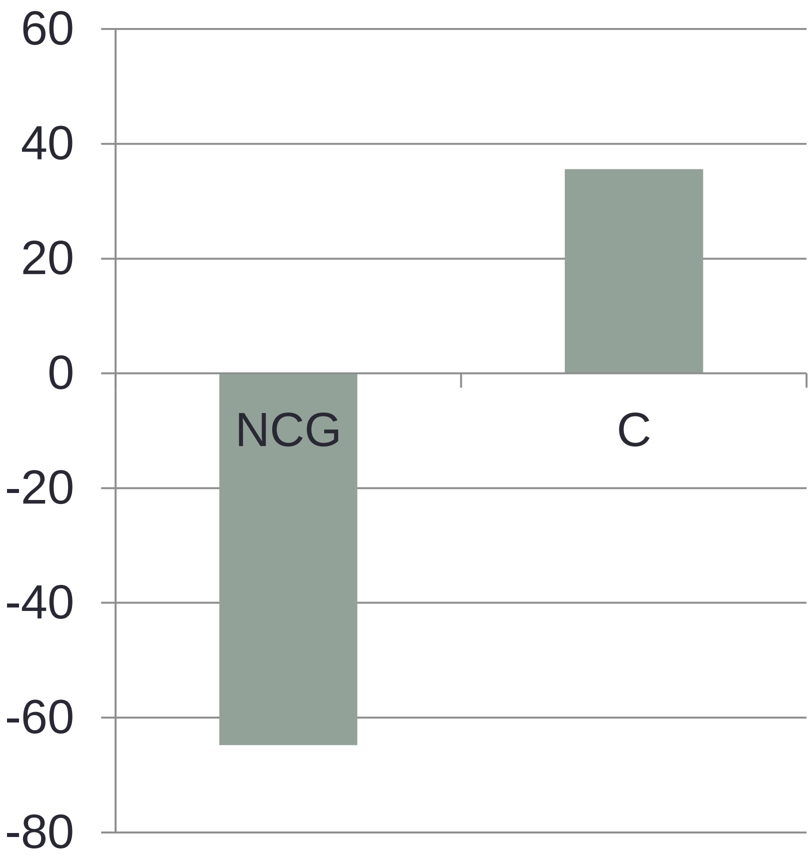
- Egalitarian benchmark suite, everything is equal
- Memory bound, little room for optimisation once at Cmm stage



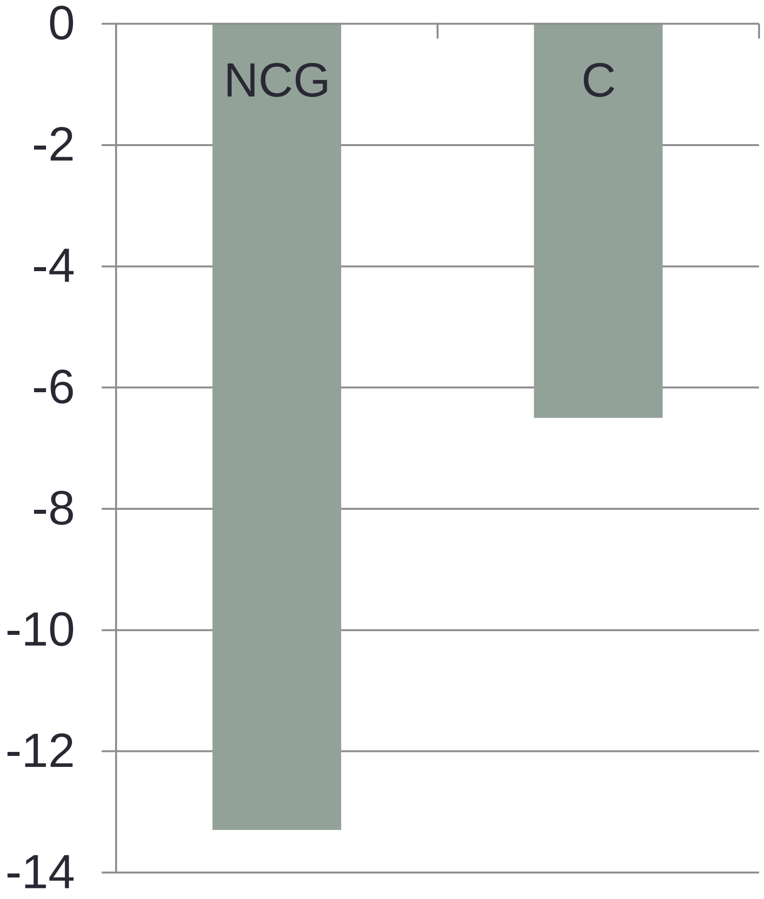
Repa Performance



Compile Times, Object Sizes



Compile Times Vs LLVM



Object Sizes Vs LLVM

Result

- LLVM Backend is simpler.
- LLVM Backend is as fast or faster.
- LLVM developers now work for GHC!

Get It

- LLVM
  - Our calling convention has been accepted upstream!
  - Included in LLVM since version 2.7

http://llvm.org

- GHC
  - In HEAD
  - Should be released in GHC 7.0

- Send me any programs that are slower!

Questions?

Why from Cmm?

A lot less work then from STG/Core

But…

Couldn’t you do a better job from STG/Core?

Doubtful…

Easier to fix any deficiencies in Cmm representation and code generator

Dealing with SSA

LLVM language is SSA form:

- Each variable can only be assigned to once
- Immutable

How do we handle converting mutable Cmm variables?

- Allocate a stack slot for each Cmm variable
- Use load and stores for reads and writes
- Use ‘**mem2reg**’ llvm optimisation pass
  - This converts our stack allocation to LLVM variables instead that properly obeys the SSA requirement

Type Systems?

LLVM language has a fairly high level type system

- Strings, Arrays, Pointers…

When combined with SSA form, great for development

- **15** bug fixes required after backend finished to get test suite to pass
- **10** of those were motivated by type system errors
- Some could have been fairly difficult (e.g returning pointer instead of value)