ErLLVM: An LLVM Backend for Erlang

Kostis Sagonas[1,2] Chris Stavrakakis[2] Yiannis Tsiouris[2]
erllvm@softlab.ntua.gr

[1]Programming Language Group, Uppsala University
[2]Software Engineering Laboratory, National Technical University of Athens

http://erllvm.softlab.ntua.gr/

September 14, 2012

- The native code compiler of Erlang
- Is mature and robust
  - Integrated in Erlang/OTP since 2001
- Produces reasonably efficient code
- Provides backends for:
  - ARM
  - SPARC V8+
  - x86 and x86_64 (AMD64)
  - PowerPC 32/64

LLVM Compiler Infrastructure

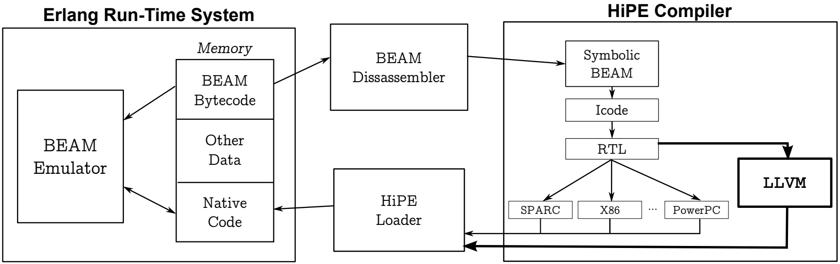
- A state-of-the-art compiler library
- Open-source with a BSD-like license
- Produces very efficient code
- Provides backends for:
  - ARMv8 32/64 and Thumb
  - SPARC V9
  - x86 and x86_64
  - PowerPC 32/64
  - Alpha
  - MIPS 32/64
  - STI CBEA Cell SPU
  - ...

A project aiming at incorporating the LLVM into the HiPE pipeline

Why use LLVM?

- Curiosity
- Easier maintenance of HiPE's code base
  - One instead of six backends
  - Parts of implementation and further optimization are "outsourced" to a community with many contributors (industry, research groups, individuals)
- More supported architectures "for free"
- Better performance
  - Target-related optimizations

LLVM backend in Erlang/OTP



- Takes as input RTL (exactly as the other HiPE backends)
  - RTL is "low-level" Erlang, yet *target-independent*
  - Erlang's high-level characteristics have been lowered
- Produces ERTS ABI-compatible code

HiPE

- Uses specific registers for arguments and return values
- Places `N` arguments in registers
- Specifies its caller-/callee-save registers
- Expects the callee to always pop the arguments (for proper tail-call support)

LLVM

- Supports several calling conventions but not HiPE's

ErLLVM

- Implements a new calling convention in LLVM (`cc11`) (**submitted a patch to LLVM team**)

HiPE

- Uses specific registers for arguments and return values
- Places `N` arguments in registers
- Specifies its caller-/callee-save registers
- Expects the callee to always pop the arguments (for proper tail-call support)

LLVM

- Supports several calling conventions but not HiPE's

ErLLVM

- Implements a new calling convention in LLVM (`cc11`) (**submitted a patch to LLVM team**)

HiPE

- Defines registers with "special" use, pinned to hardware registers (unallocatable)

VM	x86	x86_64
Native Stack Pointer	%esp	%nsp
Heap Pointer	%ebp	%r15
Process Pointer	%esi	%rbp

LLVM

- Does not provide hooks for register allocation

ErLLVM

- Translates each function definition to a new one

```
define f (%arg1) {
  ...
  res = call g (%arg1, %tmp);
  ...
  return 0;
}
```

HiPE

- Defines registers with "special" use, pinned to hardware registers (unallocatable)

VM	x86	x86_64
Native Stack Pointer	%esp	%nsp
Heap Pointer	%ebp	%r15
Process Pointer	%esi	%rbp

LLVM

- Does not provide hooks for register allocation

ErLLVM

- Translates each function definition to a new one

```
define f (%arg1) {
  ...
  res = call g (%arg1, %tmp);
  ...
  return 0;
}
```

HiPE

- Defines registers with "special" use, pinned to hardware registers (unallocatable)

VM	x86	x86_64
Native Stack Pointer	%esp	%nsp
Heap Pointer	%ebp	%r15
Process Pointer	%esi	%rbp

LLVM

- Does not provide hooks for register allocation

ErLLVM

- Translates each function definition to a new one

```
define cc11 f (%HP, %P, %arg1) {
  ...
  {%HP', %P', res} = call cc11 g (%HP, %P, %arg1, %tmp);
  ...
  return {%HP', %P', 0};
}
```

HiPE

- Prepends code to each function to handle stack overflows

LLVM

- Has a fixed Prologue/Epilogue Insertion (PEI) pass

ErLLVM

- Slightly modifies PEI pass to add HiPE-specific code to function prologue when needed (**submitted a patch to LLVM team**)

HiPE

- Prepends code to each function to handle stack overflows

LLVM

- Has a fixed Prologue/Epilogue Insertion (PEI) pass

ErLLVM

- Slightly modifies PEI pass to add HiPE-specific code to function prologue when needed (**submitted a patch to LLVM team**)

HiPE

- Provides information about the caller's frame at call sites
  - Exception handler
  - Frame size
  - Stack arity
  - Live words in frame
  - Return address of call site

LLVM

- Provides first-class support for exception handling
- Provides garbage collection intrinsics and a framework for compile-time code generation plugins

ErLLVM

- Exports information about exception handlers in the object file
- Exports garbage collection information in the object file (**submitted a patch to LLVM team**)
- Extracts all necessary information from the generated object file and creates a loadable Erlang term

HiPE

- Provides information about the caller's frame at call sites
  - Exception handler
  - Frame size
  - Stack arity
  - Live words in frame
  - Return address of call site

LLVM

- Provides first-class support for exception handling
- Provides garbage collection intrinsics and a framework for compile-time code generation plugins

ErLLVM

- Exports information about exception handlers in the object file
- Exports garbage collection information in the object file (**submitted a patch to LLVM team**)
- Extracts all necessary information from the generated object file and creates a loadable Erlang term

"Accurate Garbage Collection with LLVM" by providing...

- a framework to generate code consistent with the corresponding runtime
- GC intrinsics to mark all places that hold live pointer variables at run-time

But...

`llvm.gcroot`

"*The* `llvm.gcroot` *intrinsic is used to inform LLVM that a stack variable references an object on the heap and is to be tracked for garbage collection.*"

Big problem!

- Every root has to be placed on the stack
- Extra liveness analysis is needed for reducing stack usage

"Accurate Garbage Collection with LLVM" by providing...

- a framework to generate code consistent with the corresponding runtime
- GC intrinsics to mark all places that hold live pointer variables at run-time

But...

`llvm.gcroot`

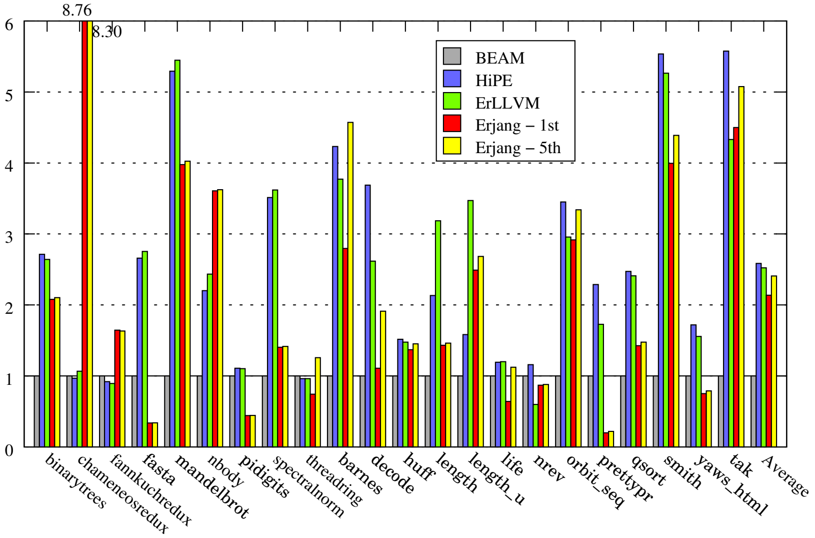
"*The* `llvm.gcroot` *intrinsic is used to inform LLVM that a stack variable references an object on the heap and is to be tracked for garbage collection.*"

Big problem!

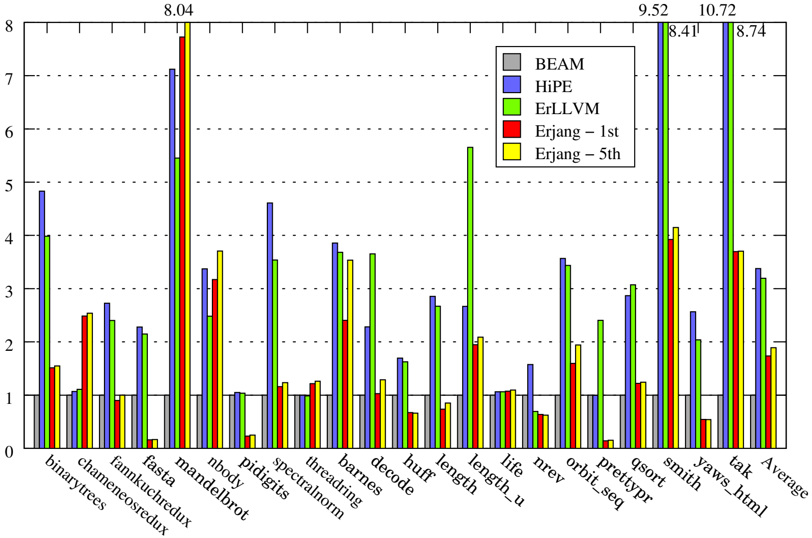
- Every root has to be placed on the stack
- Extra liveness analysis is needed for reducing stack usage

Backend	Size (LOC)			
	Code	*Blank*	*Comments*	**Total**
ARM	3886	636	830 (17.6%)	5352
SPARC	3616	643	878 (19.5%)	5137
x86/x86_64	7424	1056	1983 (21.1%)	10463
PPC32/PPC64	5001	792	891 (15.1%)	6684
LLVM (x86/x86_64)	3441	439	944 (21.5%)	4824

Runtime Performance - x86_64



Runtime Performance - x86



Pros:

+ Complete & robust: Compiles *all* Erlang programs (currently only on x86 and x86_64)
+ Fully compatible with HiPE Application Binary Interface (ABI) ⇒ Supports all Erlang features (e.g. hot-code loading, garbage collection, exception handling).
+ Smaller and simpler code base
+ Almost as fast as HiPE
+ LLVM developers now work for HiPE!

Cons:

– Suboptimal code because of LLVM's GC infrastructure
– More complicated distribution and installation
– Higher compilation times and bigger binaries

- Work on pushing LLVM and HiPE patches upstream!
- Take advantage of LLVM's features, such as the Type-Based Alias Analysis (TBAA) and the use of branch probabilities for better block placement
- Experiment with intra-module optimizations (e.g., inlining)
- Use LLVM bindings ⇒ faster compilation
- Extend the LLVM backend to support all six architectures that HiPE currently supports (e.g., ARM)
- Push for a decent LLVM GC infrastructure

Thank you!

Spam #1: Extending ErLLVM

Extensions to support currently unsupported HiPE architectures

- Add HiPE's calling convention in LLVM
- Modify PEI pass to emit HiPE-specific prologue code
- Extend `hipe_rtl2llvm` with target-specific details

Getting *more* backends

- Extend Elang Run-Time System

Spam #2: Binary Code Sizes & Compilation Times

Benchmark suite: the Standard Library (`stdlib`) and the HiPE compiler (`hipe`); comprised of 79 and 196 modules resp.

	HiPE	**ErLLVM**	**HiPE/ErLLVM**
Code Size (B)	5504880	6625368	0.83
Compilation Time (sec)	427.29	547.89	0.78

(a) `x86`

	HiPE	**ErLLVM**	**HiPE/ErLLVM**
Code Size (B)	6607584	7915928	0.84
Compilation Time (sec)	497.64	541.70	0.92

(b) `x86_64`

Spam #3: A GC example

LLVM code for handling a GC root:

```
fun foo(arg0) { ;;arg0 is root
   ...
   x <- arg0+1; ;; Last use of arg0
   ...
}
```

```
Entry:
    ;; In the entry block of the function,
    ;; allocate stack space for virtual register %X.
    %X = alloca i64*

    ;; Tell LLVM that the stack space is a stack root.
    %tmp = bitcast i64** %X to i8**
    call void @llvm.gcroot(i8** %X, i8* null)
    ;; Store the 'nil' value into it, to indicate that
    ;; the value is not live yet. "-5" is the tagged
    ;; representation of 'nil'.
    store %i64 -5, %64** %X
    ...
    ;; "CodeBlock" is the block corresponding to the
    ;; start of the scope of the virtual register %X.
CodeBlock:
    store i64 %some_value, i64** %X
    ...
    ;; As the pointer goes out of scope, store
    ;; the 'nil' value into it, to indicate that
    ;; the value is no longer live.
    store %i64 -5, %64** %X
```