ErLLVM: An LLVM back-end for HiPE, the native code compiler of Erlang/OTP

Design and Implementation

Chris Stavrakakis, Yiannis Tsiouris
{cstav,gtsiour}@softlab.ntua.gr

Software Engineering Laboratory
Division of Computer Science
Department of Electrical and Computer Engineering
National Technical University of Athens

November 5, 2011

Overview

What is ErLLVM? *(except for a cool name :-)*

ErLLVM

A project aiming at providing multiple back ends for *High Performance Erlang* (HiPE) with the use of the *Low Level Virtual Machine* (LLVM) compiler infrastructure. Ultimate goal: improve *performance* and *code maintenance*.

Outline

1. Motivation
2. Design
   - Compiler Architecture
   - Integration with ERTS
3. Evaluation
   - Complexity
   - Performance
4. Conclusion

Table of Contents

High Performance Erlang (HiPE)

- A native code compiler for Erlang.
- A project, that started at the Department of Information Technology (division of Computer Science) of Uppsala University, aimed at efficiently implementing concurrent programming systems using message-passing in general and Erlang in particular.
- Integrated in Ericsson's Open Source Erlang/OTP system since 2001.
- A *mature* project that has been developed and widely used for more than 10 years.
- Provides back ends for ARM, SPARC V8+, X86, AMD64, PowerPC and PowerPC64.

What is LLVM?

Collection of industrial strength compiler technology

- Language-independent **optimizer** and **code generator**
  Many optimizations, many targets, generates great code.
- Clang C/C++/Objective-C front end
  Designed for speed, reusability, compatibility with GCC quirks.
- Debuggers, "binutils", standard libraries
  Providing pieces of low-level toolchain, with many advantages.

Strong Point: High-level portable LLVM assembly

- ◇ RISC-like instruction set
- ◇ strict type system
- ◇ Static Single Assignment (SSA) form
- ◇ three different forms (human-readable, on-disk, in-memory)

Why LLVM?

- Used as static or just-in-time compiler, and for static code analysis.
- State-of-the-art software in C++ with a ***very*** active community of developers.
- A new compiler = glue code plus any components not yet available. Allows choice of the right component for the job, e.g. register allocator, scheduler, optimization order.
- Supports many system architectures, e.g. X86, ARM, PowerPC, SPARC, Alpha, MIPS, Blackfin, CellSPU, MBlaze, MSP430, XCore and many more!
- Open-source with a *BSD-like License* and many contributors (industry, research groups, individuals).

Similar projects

Lots of other applications:

- OpenCL: a GPGPU language, with most vendors using LLVM
- Dynamic Languages: Unladen Swallow, Rubinious, MacRuby
- llvm-gcc 4.2 and DragonEgg
- Cray Cascade Fortran Compiler
- vmkit: Java and .NET VMs
- Haskell, Mono, LDC, Pure, Roadsend PHP, RealBasic
- IOQuake3 for real-time raytracing of Quake!

`http://llvm.org/Users.html`

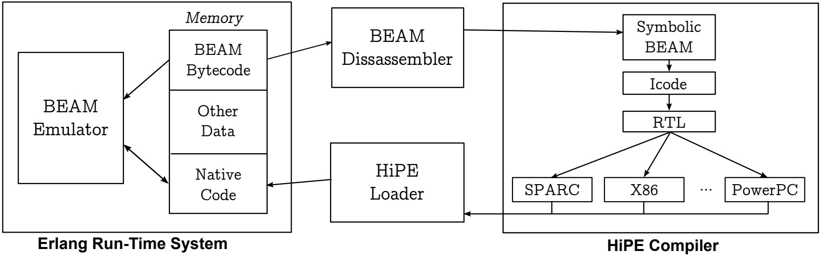
Incentive

- Simplify
  - One back end instead of N.
  - Small-sized, straightforward code.
  - Easy maintenance.
  - Outsource work on implementing and maintaining back ends!
- Performance
  - Improve run-time.

Table of Contents

HiPE's Compilation Pipeline

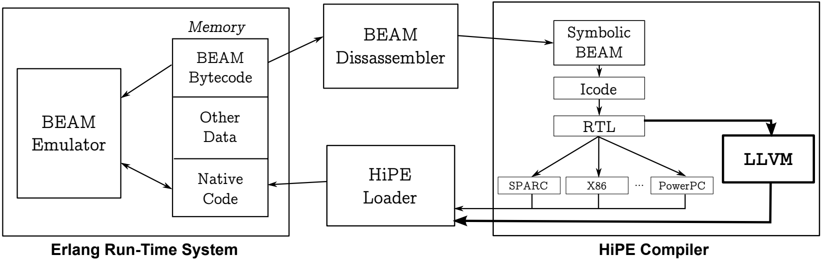
Existing pipeline:



- IR transformations:
  BEAM → Icode → RTL → Symbolic target-specific assembly
- Register allocation
- Frame management: Check for stack overflow, set-up frame, create stack descriptors, add "special" code for tail-calls.
- Linearization
- Assembler

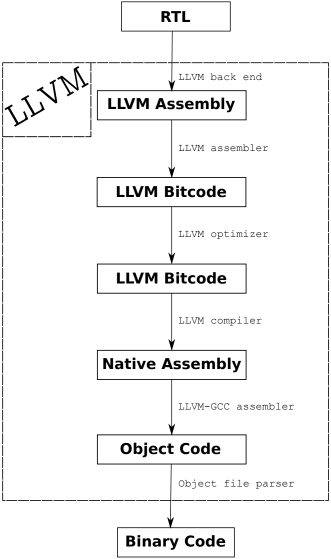
New HiPE's Compilation Pipeline

Modified pipeline:



- Place back end along with the other HiPE back ends: **after RTL**.
- RTL is low-level Erlang, yet *target-independent*.
- Erlang's high-level characteristics have been lowered.
- Use existing HiPE Loader for ERTS integration ⇒ Be ABI compatible!

The LLVM component



`hipe_rtl2llvm`	Create human-readable LLVM assembly (.ll)
`llvm-as`	Human-readable assembly (.ll) → LLVM bitcode (.bc)
`opt`	Optimization *Passes*, supports standard groups (-O1, -O2, -O3) (.bc → .bc)
llc	Bitcode (.bc) → Native assembly (.s), impose rules about memory model, stack alignment, etc.
llvm-gcc	Create object file (.s → .o)
elf64_format	Extract executable code and relocations

Subtle Points

Current work focused on providing an **AMD64** back end.

- Calling convention:
  VM “special” registers, arguments and return values,
  callee-/caller-saved registers, callee pops arguments
- Explicit frame management:
  In-lined code for stack-overflow checks in assembly prologue
- Stack descriptors:
  Exception Handling, *precise* Garbage Collection

Calling Convention

- Virtual registers with "special" use, *pinned* to hardware registers (**unallocatable**).

VM Register	AMD64 Register
Native stack pointer	`%nsp`
Heap pointer	`%r15`
Process pointer	`%rbp`

- Arguments and return values use target-specific registers.
- `NR_ARG_REGS` arguments are placed in registers.
- Certain registers of the register set are caller-/callee-save.
- Callee should *always* pop the arguments (to properly support tail calls).

$\Downarrow$ *LLVM*

LLVM handles these by implementing a custom calling convention.

Calling Convention

- Virtual registers with "special" use, *pinned* to hardware registers (**unallocatable**).

VM Register	AMD64 Register
Native stack pointer	`%nsp`
Heap pointer	`%r15`
Process pointer	`%rbp`

- Arguments and return values use target-specific registers.
- `NR_ARG_REGS` arguments are placed in registers.
- Certain registers of the register set are caller-/callee-save.
- Callee should *always* pop the arguments (to properly support tail calls).

⇓ *LLVM*

LLVM handles these by implementing a custom calling convention.

`%%XXX`: Defining caller-saved registers involved a *hack* in the Code Generator!

Calling Convention & Register Pinning

Translate each call to a new call.

- M parameters → $N + M$ parameters
- K return values → $N + K$ return values
- Correct values on function entrance and return.
- Manually scratch registers that are *no* longer needed.

VM Register	AMD64 Register
Native stack pointer	`%nsp`
Heap pointer	`%r15`
Process pointer	`%rbp`

↓

```
define f (arg1) {
  ...
  call g (arg1, arg2);
  ...
  return 0;
}
```

Calling Convention & Register Pinning

Translate each call to a new call.

- M parameters → $N + M$ parameters
- K return values → $N + K$ return values
- Correct values on function entrance and return.
- Manually scratch registers that are *no* longer needed.

VM Register	AMD64 Register
Native stack pointer	`%nsp`
Heap pointer	`%r15`
Process pointer	`%rbp`

↓

```
define hipe_cc f (NSP, HP, P, arg1) {
  ...
  call hipe_cc g (NSP', HP', P', arg1, arg2);
  ...
  return {NSP'', HP'', P'', 0};
}
```

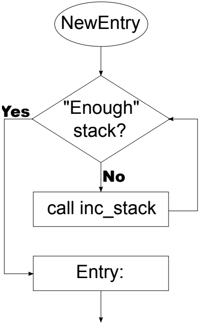
Custom Prologue

Frame management phase in HiPE's pipeline is responsible for setting-up the frame and adding stack overflow checks.

⇓ *LLVM*

Modify *(hack!)* Code Generator and add prologue code to handle stack overflow.

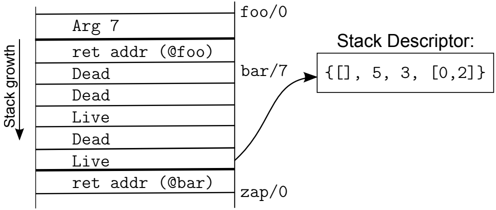
1. Start with small fixed stack.
2. If allocated stack is *not* enough (i.e. maximum frame size that might need for temps, call frames etc.), double stack frame.
3. Check again.



Stack Descriptors

Provide information about the **caller**'s frame at call sites.

- Exception handler
- Fixed frame size (excluding incoming arguments)
- Stack arity
- Live words in frame
- Return address of call site



⇓ *LLVM*

Create GC plugin in LLVM to write GC information in object file. Use `elf64_format` to parse generated object file and extract corresponding information.

Accurate Garbage Collection with LLVM

- Framework for compile time code generation *plugins* ⇒ Generate code confronting to the binary interface specified by the runtime.
- GC intrinsics to locate all places that hold live pointer variables at run-time.

`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."*

Problem: "Root property" is *not* a characteristic of a value but of a stack slot. It is responsibility of the front end to mark them as *not live* when variables that "inhabit" them are no longer live.

An example

```
{
  // A null-initialized reference to an object
  Object X;
  ...
}
```

An example

```
{
  // A null-initialized reference to an object
  Object X;
  ...
}
```

```
Entry:
;; In the entry block for the function,
;; allocate the stack space for X, which
;; is an LLVM pointer.
%X = alloca %Object*

;; Tell LLVM that the stack space is a stack
;; root. Java has type-tags on objects, so we
;; pass null as metadata.
%tmp = bitcast %Object** %X to i8**
call void @llvm.gcroot(i8** %X, i8* null)
...

;; "CodeBlock" is the block corresponding to
;; the start of the scope above.
CodeBlock:
;; Java null-initializes pointers.
store %Object* null, %Object** %X


...

;; As the pointer goes out of scope, store a
;; null value into it, to indicate that the
;; value is no longer live.
store %Object* null, %Object** %X
...
```

An example

```
{
  // A null-initialized reference to an object
  Object X;
  ...
}
```

```
Entry:
;; In the entry block for the function,
;; allocate the stack space for X, which
;; is an LLVM pointer.
%X = alloca %Object*

;; Tell LLVM that the stack space is a stack
;; root. Java has type-tags on objects, so we
;; pass null as metadata.
%tmp = bitcast %Object** %X to i8**
call void @llvm.gcroot(i8** %X, i8* null)
...

;; "CodeBlock" is the block corresponding to
;; the start of the scope above.
CodeBlock:
;; Java null-initializes pointers.
store %Object* null, %Object** %X

...

;; As the pointer goes out of scope, store a
;; null value into it, to indicate that the
;; value is no longer live.
store %Object* null, %Object** %X
...
```

Bad code is *Bad!* ←

Table of Contents

Simplify!

Back end	Size		
ARM	Total:	5362	
	Code:	3891	
	Comments:	883	(17.6%)
SPARC	Total:	5148	
	Code:	3622	
	Comments:	881	(19.6%)
X86/AMD64	Total:	10474	
	Code:	7463	
	Comments:	1953	(18.6%)
PPC/PPC64	Total:	6695	
	Code:	5009	
	Comments:	892	(15.1%)
LLVM	Total:	5288	
	Code:	3408	
	Comments:	1293	(27.5%)

LLVM

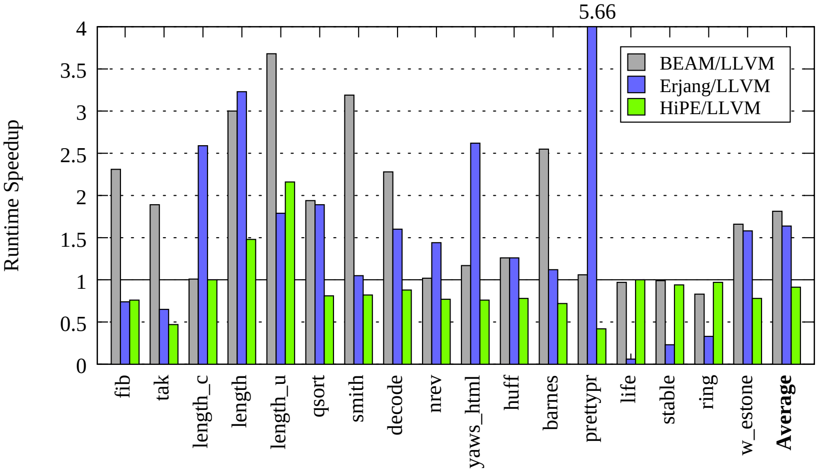
- Straightforward translation from RTL to LLVM
- $\sim 1/4$ of code is comments
- $\sim 1/4$ is the representation of LLVM language
- $\sim 1/3$ is the Object file parser module

Other

- A lot of target-specific code
- Nasty code of an assembler
- Re-inventing the wheel!

Run-time

Benchmark suite: 13 *sequential*/4 *concurrent*. 16-core Intel Xeon E7340 @ 2.40GHz/16GB RAM, Debian GNU/Linux 64-bit.



Compile Times, Object Sizes

Benchmark suite: `stdlib` (79 modules) and `hipe` (196 modules)

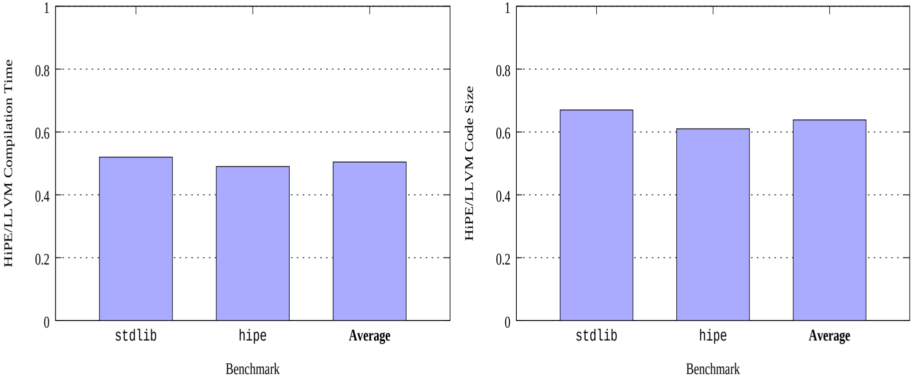


Table of Contents

Concluding Remarks

Pros:

- \+ Complete: Compiles *all* Erlang programs.
- \+ Fully compatible with HiPE Application Binary Interface (ABI). Thus, supports all Erlang features (e.g. hot-code loading, garbage collection, exception handling).
- \+ Smaller and simpler code base.
- \+ LLVM developers now work for HiPE!

Cons:

- \- Inefficient code because of LLVM's Garbage Collection infrastructure.
- \- More complicated distribution and installation.
- \- Higher compilation times.
- \- Bigger binaries.

Future Work

- Create `http://erllvm.softlab.ntua.gr` and add design and implementation technical details.
- Extend the LLVM back end to support all six architectures that HiPE currently supports.
- Improve LLVM Garbage Collection [1].
- Improve compilation times: study other ways of printing assembly (e.g. use of buffers), use Erlang LLVM bindings [2].
- Work on pushing LLVM and HiPE patches upstream!
- Provide more back ends to HiPE by extending the Erlang Run-Time System (ERTS).

Get it!

Guinea pigs are welcome! :-)

1. Grab code from ***Github***:
   i. LLVM [3]
   ii. Erlang/OTP [4]
2. Install following the instructions included in the repositories.
3. Test and measure!

Any questions?

Any questions?

Thanks!

Any questions?

Thanks!

http://dannybrown.me/wp-content/uploads/2011/01/success_baby.jpg

[1] *LLVMdev mailing list "Improving Garbage Collection" discussion.* `http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-July/041290.html`.

[2] *`llevm` is an erlang wrapper to the C API functions of LLVM created by Lukas Larsson.* `http://www.github.com/garazdawi/llevm`.

[3] *Custom LLVM implementing a HiPE ABI-compliant back end.* `http://github.com/yiannist/llvm`.

[4] *Erlang/OTP fork in order to work on implementing an LLVM back end for HiPE.* `http://github.com/yiannist/otp`.