# Just-in-time in No Time? "Use the Source!"

## How to distill a JIT compiler from an interpreter

Frej Drejhammar
<frej@sics.se>

120529

SICS

# Who am I?

- Senior researcher at the Swedish Institute of Computer Science (SICS) working on programming tools and distributed systems.
- Used to be hard-core C programmer until introduced to Scheme during undergraduate education.
- Started using Erlang "for real" while working in the same lab as Joe at SICS.

# What this talk is About

An in-depth tour of the current JIT-compiling beam emulator prototype (BEAMJIT) and the techniques used to build it.

# Outline

# Just-In-Time (JIT) Compilation

- Compile what you are about to execute to native code, at run-time.
- Fairly common implementation technique
  - Python (Psyco, PyPy)
  - Smalltalk (Cog)
  - Java (HotSpot)
  - JavaScript (SquirrelFish Extreme, SpiderMonkey)

# JIT Strategies: When to Compile

Sub-optimal to spend time to compile and optimize code which is only executed once.

- Heuristics, perhaps with help from compiler
- Run-time profiling

# JIT Strategies: What to Compile

Trade-off: Byte-code often more compact than native code.
Decide what to compile:

- Module at a time
- Method / function at a time
- A trace – a single execution path

# Tracing

Use light-weight profiling to detect when we are at a place which is frequently executed. Trace the flow of execution until we get back to the same place.

The initial strategy selected for BEAMJIT.

# Tracing: Details

- Assumes that the most likely execution path is the one we are following.
- The trace is aborted if it becomes too long.
- When we are back to the starting point: Compile.

# HiPE vs JIT

Why would Erlang need a JIT-compiler, we already have HiPE?

- Cross module optimization.
- Native-code much larger than BEAM-code.
- Tracing does not require switching to full emulation.
- Modules are target independent, simplifies deployment:
  - No need for cross compilation.
  - Binaries not strongly coupled to a particular build of the emulator.

# Outline

# BEAM: Specification

- BEAM is the name of the current VM.
- A register machine.
- Approximately 150 instructions which are specialized to approximately 450 macro-instructions using a peephole optimizer during code loading.
- No authoritative description of the semantics of the VM except the implementation source code!

# BEAM: Implementation

Fairly standard directly threaded code interpreter.

```
while (1) {
  Instr* PC;

   ...

 opcode_0: {
    /* Do something */
    PC += 3; /* Skip past immediates */
    goto **PC;
    }

 opcode_1: ...

}
```

Opcodes are addresses to code implementing that opcode.

# Outline

- Background
  - Just-In-Time Compilation
  - JIT Strategies
  - Tracing JIT
  - HiPE vs JIT
- The BEAM Emulator
  - BEAM: Specification & Implementation
- The BEAM JIT Prototype
  - Overview
  - Tools
  - Running Example
  - Source Code Representation
  - Profiling
  - Tracing
  - Code Generation
  - Executing Native Code
  - Current Status
  - Future Work
- Q&A

# Goals

- Do as little manual work as possible.
- Preserve the semantics of plain BEAM.
- Automatically stay in sync with the plain BEAM, i.e. if bugs are fixed in the interpreter the JIT should not have to be modified manually.
- Have a native code generator which is state-of-the-art.

# Plan

Our plan:

- Parse and extract semantics from the C implementation.
- Transform the parsed C source to C fragments which are then reassembled into a replacement interpreter which includes a JIT-compiler.

Compare this to other languages:

- Python's PyPy
- Smalltalk's Cog

# Tools

- LLVM – A Compiler Infrastructure, contains a collection of modular and reusable compiler and toolchain technologies. Uses a low-level assembler-like representation called IR.
- Clang – A mostly gcc-compatible front-end for C-like languages, produces LLVM-IR.
- libclang – A C library built on top of Clang, allows the AST of a parsed C-module to be accessed and traversed.

# What do we need?

- A way to profile.
- A way to represent the emulator source code.
- A way to trace execution.
- A way to convert a trace into native code.
- A way to share emulator state between interpreter and native code.

All without slowing down the interpreter too much.

# Profiling

- Let the compiler insert profiling instructions at the head of loops.
- Maintain a counter and when a threshold is reached, turn on tracing.

```
loop(0) ->
    ok;
loop(N) ->
    loop(N-1).
```

```
{function, loop, 1, 2}.
  {label,1}.
    {func_info,{atom,ex},{atom,
               loop},1}.
  {label,2}.
    {jit_profile,0}.
    {test,is_eq_exact,{f,3},
       [{x,0},{integer,0}]}.
    {move,{atom,ok},{x,0}}.
    return.
  {label,3}.
    {gc_bif,'-',{f,0},1,
      [{x,0},{integer,1}],{x,0}}
                 .
    {call_only,1,{f,2}}.
```

# Example

```
loop(0) ->
    ok;
loop(N) ->
    loop(N-1).
```

```
{function, loop, 1, 2}.
  {label,1}.
    {func_info,{atom,ex},{atom,loop},1}.
  {label,2}.
    {jit_profile,0}.
    {test,is_eq_exact,{f,3},[{x,0},{integer,0}]}
              .
    {move,{atom,ok},{x,0}}.
    return.
  {label,3}.
    {gc_bif,'-',{f,0},1,
      [{x,0},{integer,1}],{x,0}}.
    {call_only,1,{f,2}}.
```

# Implementation

```
_op_i_is_eq_exact_immed_frc: {
  Instr* next = (Instr *) *(I + 2 + 1);
  if (x0 != I[(1)+1]) {
    goto _op_jump_f;
  }
  I += 2 + 1;
  goto *(beam_ops[(Instr)next]);
}

_op_jump_f: {
  I = ((Instr *) I[(0)+1]);
  goto *(beam_ops[(BeamInstr)*I]);
}
```

# Source Code Representation

Preliminary step: Parse and simplify emulator source

- Flatten variable scopes
- No fall-troughs
- Remove loops, replace by if+goto
- Turn structured C into a spaghetti of **B**asic **B**locks (BB), CFG – Control Flow Graph.
- Do liveness-analysis of variables.

```
_op_i_is_eq_exact_immed_frc: {
  Instr* next = (Instr *) *(I + 2 + 1);
  if (x0 != I[(1)+1]) {
    goto _op_jump_f;
  }
  I += 2 + 1;
  goto *(beam_ops[(Instr)next]);
}

_op_jump_f: {
  I = ((Instr *) I[(0)+1]);
  goto *(beam_ops[(BeamInstr)*I]);
}
```

```
BeamInstr *next_125;
...
_op_i_is_eq_exact_immed_frc:
    jnext_125 = *(I + 3);
    if (x0 != I[2]) {
        goto _lbl_2487;
    } else {
        goto _lbl_429;
    }

_lbl_2487:
    I = (I)[1];
    goto *beam_ops[*I];

_jit_anon_lbl_429:
    I += 3;
    goto *beam_ops[next_125];
```

# Tracing: Recording Execution Flow

Exploit that the code is split into BBs, on entry record:

- Current I (Program counter)
- An identifier for the BB

```
BeamInstr *next_125;
...
_op_i_is_eq_exact_immed_frc:
    jnext_125 = *(I + 3);
    if (x0 != I[2]) {
        goto _lbl_2487;
    } else {
        goto _lbl_429;
    }

_lbl_2487:
    I = (I)[1];
    goto *beam_ops[*I];

_jit_anon_lbl_429:
    I += 3;
    goto *beam_ops[next_125];
```

```
BeamInstr *next_125;
...
_trace_op_i_is_eq_exact_immed_frc:
    if (jit_trace_append_bb(c_p->trace, I, 214)) {
        beam_ops = jit_plain_ops;
        goto _op_i_is_eq_exact_immed_frc;
    }
    _next_125 = *(I+3);
    if (x0 != I[2]) {
        goto _trace_lbl_2487;
    } else {
        goto _trace_lbl_429;
    }

_trace_lbl_2487:
    if (jit_trace_append_bb(c_p->trace, I, 745)) {}
    I = (I)[1];
    goto *beam_ops[*I];
_trace_lbl_429:
    if (jit_trace_append_bb(c_p->trace, I, 1388)) {}
    I += 3;
    goto *beam_ops[next_125];
```

# Tracing: Enabling/Disabling Tracing

- Generate **two** implementations of each opcode, a plain and a tracing version.
- Make the interpreter indirectly threaded.

```
while (1) {
  Instr* PC;

  ...

 opcode_0: {
    /* Do something */
    PC += 3; /* Skip past
                immediates */
    goto **PC;
    }

 opcode_1: ...
}
```

```
while (1) {
  Instr* PC;

  ...

 opcode_0: {
    /* Do something */
    PC += 3; /* Skip past
                immediates */
    goto *ops[*PC];
    }

 opcode_1: ...

}
```

- Switching implementation is just a matter of changing ops.
- Surprisingly small effect on performance.

# Tracing: Trace representation

Currently naive:

- One ongoing trace per process.
- Fixed maximum length.
- Only one ongoing trace starting from the same profiling instruction.
- Large potential for improvement.

# Code Generation: Introduction

- Use LLVM's optimizer and native code generator.
- LLVM understands LLVM-IR, we have C.
- Do not want to implement a C-compiler – Generate stubs which are then compiled to IR using Clang.
- JIT code generator reads IR during initialization and extracts relevant parts from the stubs.
- Trace is traversed and stub fragments are glued together, to a function implementing the trace.
- The function is then optimized and compiled to native code using LLVM.

SICS

# Code Generation: Stubs

- Stubs return the value of the conditional (for conditional branches), or the new I (for opcode dispatch).

```
BeamInstr *next_125;
...
_op_i_is_eq_exact_immed_frc:
    jnext_125 = *(I + 3);
    if (x0 != I[2]) {
        goto _lbl_2487;
    } else {
        goto _lbl_429;
    }

_lbl_2487:
    I = I[1];
    goto *beam_ops[*I];

_jit_anon_lbl_429:
    I += 3;
    goto *beam_ops[next_125];
```

```
int op_i_is_eq_exact_immed_frc(void)
{
  BeamInstr *I;
  BeamInstr *next_125;
  Eterm x0;

  next_125 = *(I + 3);
  return (x0 != I[2]) != 0;
}

void *lbl_2487(void)
{
  BeamInstr *I;
  I = I[1];
  return (void*)*I;
}

void *lbl_429(void)
{
  BeamInstr *I;
  BeamInstr *next_125;
  I += 3;
  return (void*)next_125;
}
```

SICS

# Code Generation: Resulting Function

When two BBs are linked to each other through a conditional branch:

- Insert conditional that checks that we still are on the fast-path.
- If we are on the slow path, return which BB execution should continue from.

```
int trace_fun(void)
{
  BeamInstr *I;
  BeamInstr *next_125;
  Eterm x0;

 start:
  next_125 = *(I + 3);
  if ((x0 != I[2]) != 0)
    return 2487; /* BB */
  I += 3;

  if (I[next_125] != trace[index+1].I)
    return next_125; /* Opcode, The index of an opcode and the first BB in its
           implementation overlap */

  /* N - 1 etc */

  goto start;
}
```

Shown as C, actually done on IR.

# Code Generation: Optimizations

- Insert a `I=trace[index].I;` at the start of each opcode stub. (future work)
- Teach LLVM-optimizer that anything accessed via `I` is a compile-time constant. (future work)

```
int trace_fun(void)
{
  BeamInstr *I;
  BeamInstr *next_125;
  Eterm x0;

 start:
  next_125 = *(I + 3);
  if ((x0 != I[2]) != 0)
    return 2487; /* BB */
  I += 3;

  if (I[next_125] != trace[index+1].I)
    return next_125;

  /* N - 1 etc */

  goto start;
}
```

```
int trace_fun(void)
{
  Eterm x0;

 start:
  if (x0 != 0)
    return 2487; /* BB */

  /* N - 1 etc */

  goto start;
}
```

# Executing Native Code

- Interpreter keeps much of its state in local variables.
- Need a way to pass that state to the native code.
- Collecting state into a shared C-struct would be simple, but not efficient.
- Solution is to copy the state into a buffer which is used to initialize the locals in the native code.
- Do the opposite when we fall off the fast path.
- Use liveness information to avoid copying data we do not need.
- Generated from source.

# Current Status

- First working version a week ago today.
- Only unicore.
- Naive tracing.
- Traces are never GC:d.
- Lacking optimizations.
- Conservative liveness analysis.
- Too early to get any performance figures.

# Future Work

- Implement JIT-specific optimizations.
- Manage traces and compiled native code.
- Improve liveness analysis.
- Integrate with code update.
- Performance evaluation.
- SMP-support.
- Extend JIT:ing to BIFs.

# Outline



SICS

# Acknowledgments


- Ericsson – For funding the project and letting me do a cool hack as (part of) my job.
- Jonas Sjöbergh – For the title of the talk.

# Just-in-time in No Time? "Use the Source!"

Questions?