BEAMJIT, a Maze of Twisty Little Traces
A walk-through of the prototype just-in-time (JIT) compiler for Erlang.

Frej Drejhammar
<frej@sics.se>

130613
Who am I?

- Senior researcher at the Swedish Institute of Computer Science (SICS) working on programming tools and distributed systems.
What this talk is About

A brief introduction to the BEAM just-in-time compiler followed by a walk-through of last year’s development.
Outline

Background
- Just-In-Time Compilation
- BEAM: Specification & Implementation
- Project Goal
- Tools

JIT:ing as it applies to BEAM

The BEAM JIT Prototypes

Future Work

Acknowledgements & Questions
Just-In-Time (JIT) Compilation

- Decide at runtime to compile “hot” parts to native code.
- Fairly common implementation technique
  - Python (Psyco, PyPy)
  - Smalltalk (Cog)
  - Java (HotSpot)
  - JavaScript (SquirrelFish Extreme, SpiderMonkey)
BEAM is the name of the Erlang VM.

- A register machine.
- Approximately 150 instructions which are specialized to approximately 450 macro-instructions using a peephole optimizer during code loading.
- Hand-written C (mostly) directly threaded interpreter.
- No authoritative description of the semantics of the VM except the implementation source code!
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:
- 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.
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 stay target independent, simplifies deployment:
  - No need for cross compilation.
  - Binaries not strongly coupled to a particular build of the emulator.
**Tools**

- **LLVM** – A Compiler Infrastructure, contains a collection of modular and reusable compiler and toolchain technologies. Uses a low-level assembler-like representation called LLVM-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.
Just-In-Time (JIT) Compilation as it Applies to BEAM

- 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.
- Compile trace to native code.
- NOTE: We are tracing the execution flow in the interpreter, the granularity is finer than BEAM opcodes.
BEAMJIT: What is Needed?

- Three basic execution modes
  - Profiling
  - Tracing
  - Native

- Interpreter loop has to be modified to support mode switching:
  - Turn on/off tracing.
  - Passing state to/from native code.

- Native code generation: Need the semantics for each instruction.
Extracting the Semantics of the BEAM Opcodes

Use libclang to parse and simplify the interpreter source:

- Flatten variable scopes.
- Remove loops, replace by if + goto.
- Make fall-troughs explicit.
- Turn structured C into a spaghetti of Basic Blocks (BB), CFG – Control Flow Graph.
- Do liveness-analysis of variables.
Outline

Background
  Just-In-Time Compilation
  BEAM: Specification & Implementation
  Project Goal
  Tools

JIT:ing as it applies to BEAM

The BEAM JIT Prototypes

Future Work

Acknowledgements & Questions
BEAMJIT Evolution

Evolution since last year
- Mk. I (EUC’12)
- Mk. Ib
- MK. II
- MK. III
- Mk. IV (EUC’13)
BEAMJIT Mk. I: Profiling

- First step in figuring out what to JIT-compile
  - Let Erlang compiler insert profile instructions at places which can be the head of a loop.
  - Count the number of times a function is executed.
  - Trigger tracing when count is high enough.
  - Eventually everything is compiled, this is BAD.

- Requires implementing (by hand) the profile-instruction in the interpreter.
BEAMJIT Mk. I: Tracing

- Switch to a new version of the interpreter, generated from the CFG.
- For each basic block we pass through, record basic block identity and PC.
- Abort trace if too long.
- If we reach the profile instruction we started the trace from – We have found a loop!
Have two implementations of each opcode.
Switch the table of opcodes.
BEAMJIT Mk. I: Cost of Indirect Threading
Glue together LLVM-IR-fragments for the trace.

*Guards* are inserted to make sure we stay on the traced path.

Hand the resulting IR off to LLVM.

Fragments are extracted from the CFG as C-source, compiled to IR using clang (at build-time) and loaded during system initialization.
BEAMJIT Mk. I: Calling Native Code

- **Interpreter → Native:**
  - Interpreter: Copy live variables to a structure.
  - Native: Load vars into temporaries.

- **Native → Interpreter:**
  - The reverse.
  - Jump to the BB to continue from.
- Depressing performance.
- Running in pure interpreting mode, 6-7 times slower.

```c
x = ...;
if (x != 0)
    x = x + 1;
else
    x = x - 1;
...
```

```
regX = ...
brZ bb1, bb0
...
bb0:
    inc regX
    ...
bb1:
    inc regX
    ...
regX = ...
```
BEAMJIT Mk. Ib: First Useful

- First version that could compile OTP without crashing and pass the test suite.
- Make profiler time-aware.
- Measure execution intensity by including timestamp, count is incremented if the function was executed recently, reset otherwise.
- Blacklist locations which:
  - Never produce a successful trace.
  - Where we leave the trace without executing the loop at least once.
- GC traces when they are no longer needed.
- Minor performance improvements.
- Modify the interpreter loop as little as possible.
- Have separate trace interpreter.
- Limit entry to the interpreter at instruction boundaries.
- Have separate *cleanup*-interpreter to continue execution to the next instruction boundary.

Profiling ➔ Tracing ➔ Native ➔ Cleanup
Use liveness information from the CFG.

Package native-code as a function where the arguments are the live variables.

The cleanup-interpreter is a set of functions, one for each BB, which tail-recursively calls the next BB. Arguments are the live variables.
BEAMJIT Mk. II: Performance

- Performance not stellar.
- Sensitive to placement in source-code.
- Should be possible to optimize further.
BEAMJIT Mk. III: Trace-Along

- Appears that we quite often compile a trace which is not representative.
- Ensure that we have a representative trace: Trace-Along
  - Follow along a previously created trace.
  - Abort trace if we diverge.
  - Generate code when succeeded multiple times.
We blacklist many locations where trace-along repeatedly fails to find a representative trace.

- Allow multi-path traces.
- Generate native code when the trace has not grown for $N$ successive iterations.
- Slows down LLVM optimization and native code generation significantly.
BEAMJIT Mk. IV: Trace Compression

- LLVM slowdown appears to be related to the size of the CFG.
- Inspection of traces shows loops and common segments.
- Compress traces to remove shared segments.
BEAMJIT Mk. IV: Performance

- Compilation overhead dwarfs everything else (-O2).
- Future work: Figure out which optimization passes are needed.
GUARDS COSTLY.

NOT GOOD WHERE THE COMMON CASE CANNOT BE ON THE FAST PATH.
Future Work

- Do not fixate on finding loops
  - Allow traces which are runs rather than loops, ring benchmarks.

- Erlang-aware constant propagation:
  - Eliminate loads from code (constant at compile time).
  - Will eliminate loading of immediates.
  - Will eliminate many of the guards.

- Increase performance in plain interpreting mode.
- Run native-code generation in separate thread.
- Extend trace outside the main interpreter loop, inside BIFs.
Outline

Background
  Just-In-Time Compilation
  BEAM: Specification & Implementation
Project Goal
Tools
JIT:ing as it applies to BEAM
The BEAM JIT Prototypes
Future Work
Acknowledgements & Questions
Acknowledgements

This work is funded by Ericsson AB.
Questions?