BEAMJIT: An LLVM based just-in-time compiler for Erlang

Frej Drejhammar
<frej@sics.se>

140407

SWEDISH ICT SICS

Who am I?

Senior researcher at the Swedish Institute of Computer Science (SICS) working on programming languages, tools and distributed systems.

Acknowledgments

- Project funded by Ericsson AB.
- Joint work with Lars Rasmusson `<lra@sics.se>`.

SWEDISH ICT SICS

What this talk is About

- Automatic synthesis of a JIT-compiling VM for Erlang.
- Our experiences with LLVM and MCJIT.

Outline

SWEDISH ICT SICS

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)

Erlang

- Functional language developed by Ericsson.
- Soft real-time.
- Multi-threaded with share-nothing semantics.
- Message passing.
- Powerful supervision primitives.
- Hot code loading and replacement.
- OTP: Framework for writing fault-tolerant applications.
- Compiled to virtual machine (VM), BEAM.

BEAM: Specification & Implementation

- BEAM is the name of the Erlang VM.
- A register machine.
- Approximately 150 instructions which are specialized to around 450 macro-instructions using a peephole optimizer during code loading.
- Instructions are CISC-like.
- Hand-written C (mostly) directly threaded interpreter.
- No authoritative description of the semantics of the VM except the implementation source code!
- HiPE – a ahead-of-time native compiler
  - Traditional back-end for x86, PowerPC, SPARC, ARM
  - ErLLVM back-end based on LLVM

Motivation

- A JIT compiler increases flexibility.
- Compiled BEAM modules are platform independent.
- Cross-module optimization.
- Integrates naturally with code upgrade.

Project Goals

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.

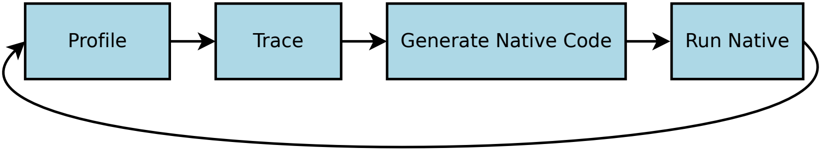
Plan:

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

Outline

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.

Profiling

- First step in figuring out what to JIT-compile
- Let Erlang compiler insert profile instructions at locations which can be the head of a loop
- Maintain a time stamp and counter for each location
- Measure execution intensity by incrementing a counter if the location was visited recently, reset otherwise
- Trigger tracing when count is high enough
- Blacklist locations which:
  - Never produce a successful trace.
  - Where we, when executing native code, leave the trace without executing the loop at least once.

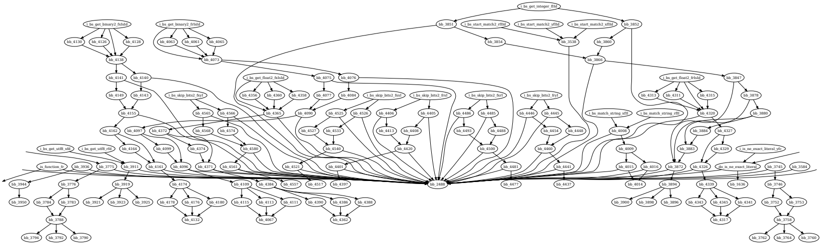
Outline

SWEDISH ICT SICS

Extracting the Semantics of the BEAM Opcodes

Use libclang to parse and simplify the interpreter source:

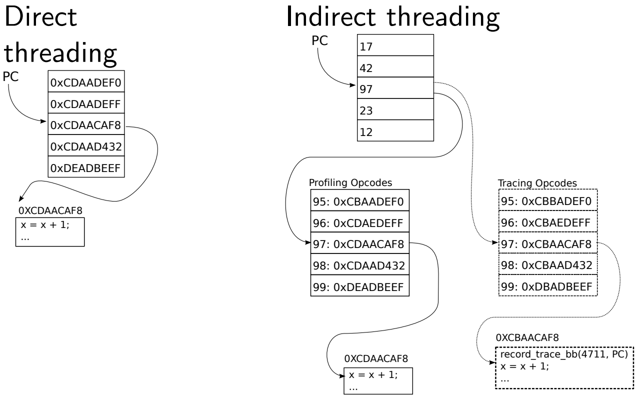
- Use Erlang binding for libclang.
- Flatten variable scopes.
- Remove loops, replace by if + goto.
- Make fall-troughs explicit.
- Turn structured C into a spaghetti of Basic Blocks (BB), CFG – Control Flow Graph.
- Do liveness-analysis of variables.



Naïve Tracing

- Use a new version of the interpreter, generated from the CFG.
- Generate a tracing and non-tracing version of each opcode.
- 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!

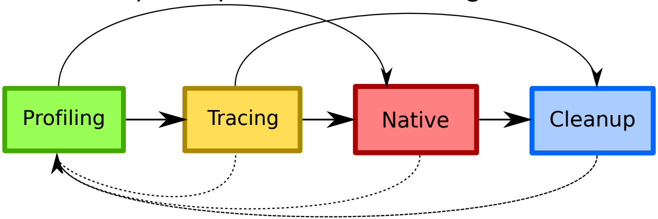
Naïve Profiling to Tracing Mode Switch



- Have two implementations of each opcode.
- Switch the table of opcodes.
- Compiler has to assume that a mode switch can take place at any block $\rightarrow$ performance suffers

Refined Tracing

- 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.
- Reuse *cleanup*-interpreter when returning from native mode.



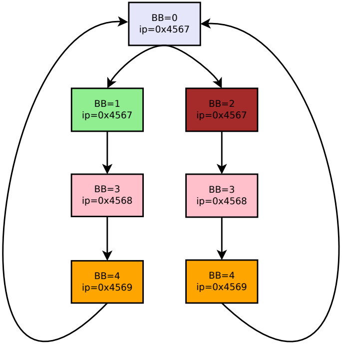
Further Tracing Refinements

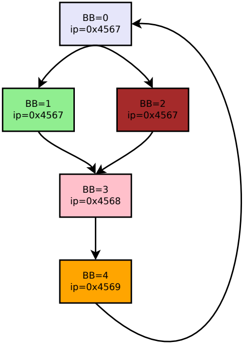
Ensure that we have a representative trace:

- Follow along a previously created trace.
- Allow multi-path traces.
- Generate native code when the trace has not grown for N successive iterations.
- Enforce limit on total size of trace.

Trace Compression

- Large CFGs slow down LLVM optimization and native code generation significantly.
- Solution: Compress traces to remove shared segments.

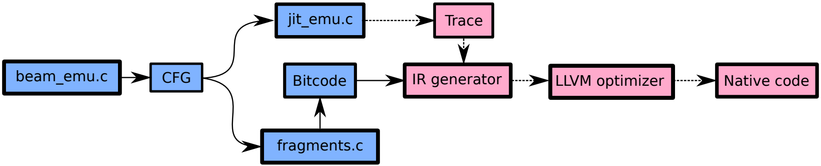




Outline

Native-code Generation

- Glue together LLVM-IR-fragments for the trace.
- *Guards* are inserted to make sure we stay on the traced path.
- Fragments are extracted from the CFG as C-source, compiled to IR using clang (at build-time) and loaded during system initialization.
- Hand the resulting IR off to LLVM for the rest.



Calling Native Code

Switching from interpreter to native code:

- Use liveness information from the CFG.
- Package native-code as a function where the arguments are the live variables.

Switching from native code to interpreter:

- The cleanup-interpreter is a set of functions, one for each BB, which tail-recursively calls the next BB. Arguments are the live variables.
- Cleanup-interpreter packs up live variables in a structure which the interpreter unpacks on return.

Performance Improvements

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

Performance

Steady state:

- Eliminates most of the instruction decode overhead.
- Up to 50% reduction in runtime.
- Well-behaved programs: around 25%.
- Programs using cute tricks such as unrolling: up to 200% increase in runtime.

Future Work

- Full SMP support.
- Box/unboxing-aware constant propagation.
- Extend JIT support to fold in primitives.

Outline

LLVM's Strengths

- Access to the C AST via libClang.
- Quality of generated code.
- The optimization framework.

LLVM's Weaknesses

- No thread-safety – Extra housekeeping to do things in the correct context.
- Compilation could be much faster.
- Native code has to be packaged as C-function – Would really like to have enough control to patch generated native code.
- Clang/LLVM does bad job on the main interpreter:
  - Allocates the VM's stack- and instruction-pointers on the stack.
  - Inserts extra instruction sequence before indirect jumps (when dispatching to next VM instruction), GCC appears to insert it at the branch target and only if needed.

  Costs us 15-20%

Outline

Questions?