A Status Update of BEAMJIT, the Just-in-Time Compiling Abstract Machine

Frej Drejhammar and Lars Rasmusson
<{frej,lra}@sics.se>

140609

Who am I?

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

Acknowledgments

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

What this talk is About

An introduction to how BEAMJIT works and a detailed look at some subtle details of its implementation.

Outline

Background

BEAMJIT from 10000m

BEAMJIT-aware Optimization

Compiler-supported Profiling

Future Work

Questions

Just-In-Time (JIT) Compilation

- Decide at runtime to compile “hot” parts to native code.
- Fairly common implementation technique.
    - McCarthy’s Lisp (1969)
    - Python (Psyco, PyPy)
    - Smalltalk (Cog)
    - Java (HotSpot)
    - JavaScript (SquirrelFish Extreme, SpiderMonkey, JägerMonkey, IonMonkey, V8)

Motivation

- A JIT compiler increases flexibility.
  - Tracing does not require switching to full emulation.
  - Cross-module optimization.
- Compiled BEAM modules are platform independent:
  - No need for cross compilation.
  - Binaries not strongly coupled to a particular build of the emulator.
- Integrates naturally with code upgrade.

Project 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.
- Eventually be better than HiPE (steady-state).

Plan

- Use automated tools to transform and extend the BEAM.
- Use an off-the-shelf optimizer and code generator.
- Implement a tracing JIT compiler.

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 (mostly) C directly threaded interpreter.
- No authoritative description of the semantics of the VM except the implementation source code!

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.

Tracing Just-in-time Compilation

Figure out the execution path in your program which is most frequently traversed:

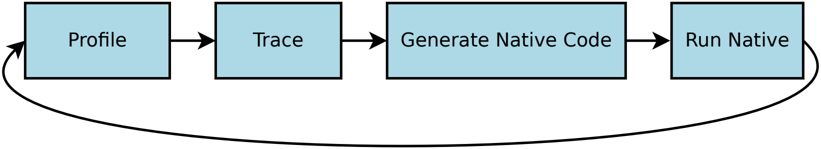
- Profile to find hot spots.
- Record the execution flow from there.
- Turn the recorded trace into native-code.
- Run the native-code.

Outline

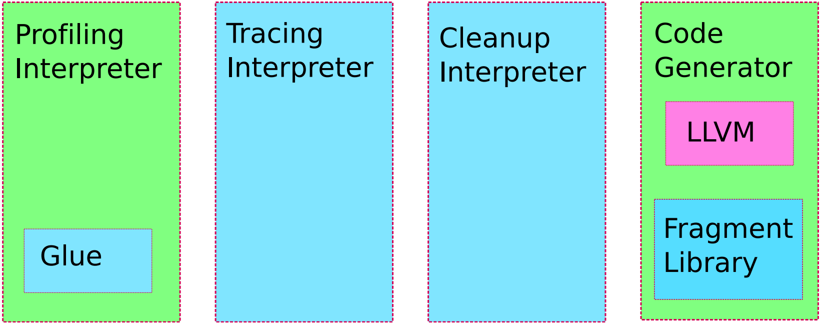
- Background
- BEAMJIT from 10000m
  - Components
  - Profiling
  - Tracing
  - Native-code Generation
  - Concurrency
  - Performance
- BEAMJIT-aware Optimization
- Compiler-supported Profiling
- Future Work
- Questions

BEAMJIT from 10000m

- Use light-weight profiling to detect when we are at a place which is frequently executed.
- Trace the flow of execution until we have a representative trace.
- Compile trace to native code.
- Monitor execution to see if the trace should be extended.



BEAMJIT from 10000m: Components



Profiling Interpreter

Glue

Tracing Interpreter

Cleanup Interpreter

Code Generator

LLVM

Fragment Library

- Blue-colored parts generated automatically by a `libClang`-based program.
- Separate interpreters result in better native-code for the different execution modes compared to a single interpreter supporting all modes.
- We have to limit the set of entry points to the profiling interpreter to preserve performance – Cleanup-interpreter executes partial BEAM-opcodes.

BEAMJIT from 10000m: Profiling

- The compiler identifies locations, *anchors*, which are likely to be the start of a frequently executed BEAM-code sequence.
- The runtime-system measures the execution intensity of each *anchor*.
- A high enough intensity triggers tracing.

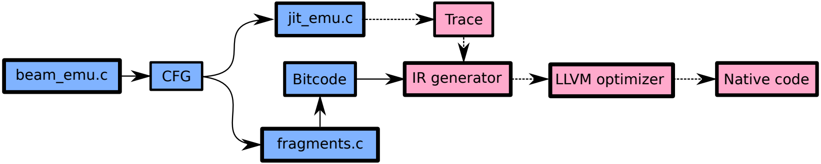
... one of the details, more later.

BEAMJIT from 10000m: Tracing

- Tracing uses a separate interpreter.
- During tracing we record the BEAM `PC` and the identity of each (interpreter) basic-block we execute.
- A trace is considered successful if:
    - We reach the *anchor* we started from.
    - We are scheduled out.
- Follow along previously recorded traces to limit memory consumption.
- Native-code generation is triggered when we have had N successive successful traces without the recorded trace growing.

BEAMJIT from 10000m: Native-code Generation

- Glue together LLVM-IR-fragments for the trace.
- Fragments are extracted from the BEAM implementation and pre-compiled to LLVM-bitcode (LLVM-IR) and loaded during BEAMJIT initialization.
- *Guards* are inserted to make sure we stay on the traced path. A failed guard results in a call to the Cleanup-interpreter.
- Hand the resulting IR off to LLVM for optimization and native-code emission.
- LLVM optimizer extended with a BEAM-aware pass (more later).



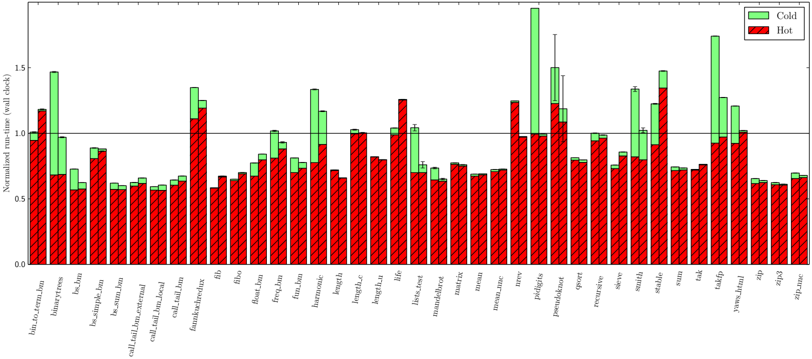
BEAMJIT from 10000m: Concurrency

- IR-generation, optimization and native-code emission runs in a separate thread.
- Tracing is disabled when compilation is ongoing.
- LLVM is slow, asynchronous compilation masks the cost of JIT-compilation.

BEAMJIT from 10000m: Performance

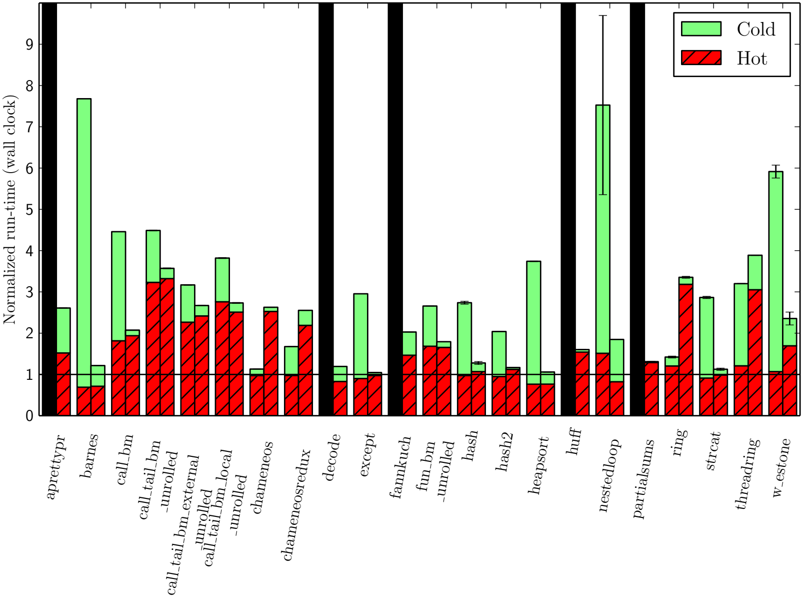
- Currently single-core (Poor-man's SMP-support started working last week).
- Currently hit or miss, although more hit than miss.
- Removes overhead for instruction decoding (more later).
- For short benchmarks tracing overhead dominates.
- Some discrepancies we have yet to explain.

Performance (Good)



- Execution time of BEAMJIT normalized to the execution time of BEAM (1.0)
- Left column: synchronous compilation
- Right column: asynchronous compilation
- Cold: no preexisting native code
- Hot: stable state

Performance (Bad)



- (Same setup as previous slide)

Outline

Background

BEAMJIT from 10000m

BEAMJIT-aware Optimization
- Optimizations in LLVM
- A hypothetical BEAM Opcode
- Optimization
- Result

Compiler-supported Profiling

Future Work

Questions

Optimizations in LLVM

- State-of-the-art optimizations.
- Surprisingly good at eliminating redundant tests etc.
- Cannot help us with a frequently occuring pattern.

A hypothetical BEAM Opcode

PC-1	...
PC	&&add_immediate
PC+1	<register-index>
PC+2	<immediate-value>
PC+3	...

```
int regs[N];
...
add_immediate:
    int reg = load(PC+1);
    int imm = load(PC+2);

    regs[reg] += imm;
    PC += 3;
    goto **PC;
```

Optimization

```
/* Previous entry */
int reg = load(PC+1);
int imm = load(PC+2);

regs[reg] += imm;
PC += 3;
/* the next entry follows */
```

- This is Erlang, the code area is constant, PC points to constant data.
- The trace stores `PC` values.
- Guards check that we are on the trace.
- Known `PC` on entry to each basic block.
- Do the loads at compile-time

Result

```
regs[1/*load(PC+1)*/]+=2/*load(PC+2)*/;
PC = 0xcab00d1e;
/* the next entry follows */
```

- The `PC`-update will most likely be optimized away too.

Outline

- Background
- BEAMJIT from 10000m
- BEAMJIT-aware Optimization
- Compiler-supported Profiling
  - Motivating Profiling
  - Profiling at Run-time
  - Where Should the Compiler Insert Anchors?
- Future Work
- Questions

Motivating Profiling

- The purpose of profiling is to find frequently executed BEAM-code to convert into native code.
- Reducing the run-time for the most frequently executed parts of a program will have the largest impact for the effort we invest.
- Traditionally inner loops are considered a good target.
- The compiler can flag loop heads – The run-time does not need to be smart.
- We call the flagged locations in the program for *anchors*.

Profiling at Run-time

- Maintain a time stamp and counter for each *anchor*.
- Measure execution intensity by incrementing a counter if the *anchor* was visited recently, reset otherwise.
- Trigger tracing when count is high enough.
- Blacklist *anchor* which:
    - Never produce a successful trace.
    - Where we, when executing native code, leave the trace without executing one path through the trace at least once.

Where Should the Compiler Insert *Anchors*?

- At the head of loops!
- Erlang does not have syntactic looping constructs.
- List-comprehensions do not count.
- To iterate is human, to recurse divine – Add an *anchor* at the head of every function.
- Is this enough?

Where Should the Compiler Insert *Anchors*?

```
mul4(N) ->
    anchor(),
    case N of
        0 -> 0;
        N -> 4 + mul4(N-1)
    end.
```

- How many loops can you see?

Where Should the Compiler Insert *Anchors*?

```
mul4(N) ->
    anchor(),
    case N of
        0 -> 0;
        N ->
            Tmp = mul4(N-1),
            anchor(),
            4 + Tmp
    end.
```

- An *anchor* is needed after each call which is not in a tail position.
- Is this enough?

Where Should the Compiler Insert *Anchors*?

Remember:

- A trace starts at an anchor and ends when:
  - We reach the *anchor* we started from.
  - We are scheduled out.
- What does this imply for an event handler?

Where Should the Compiler Insert *Anchors*?

```
handler(State) ->
    anchor(),
    receive
        {add, Arg} ->
            handler(State + Arg);
        {sub, Arg} ->
            handler(State - Arg)
    end.
```

Where Should the Compiler Insert *Anchors*?

```
handler(State) ->
    anchor(),
   M = wait_for_message(),
    case M of
        {add, Arg} ->
            handler(State + Arg);
        {sub, Arg} ->
            handler(State - Arg);
        _ ->
            postpone_delivery(M)
    end.
```

Execution path: scheduled in → do-pattern-matching → call `handler` → trigger-tracing → scheduled out.

Where Should the Compiler Insert *Anchors*?

```
handler(State) ->
    anchor(),
    M = wait_for_message(),
    anchor(),
    case M of
        {add, Arg} ->
            handler(State + Arg);
        {sub, Arg} ->
            handler(State - Arg);
        _ ->
            postpone_delivery(M)
    end.
```

Execution path: scheduled in → trigger-tracing → do-pattern-matching → call `handler` → scheduled out.

Outline

Background

BEAMJIT from 10000m

BEAMJIT-aware Optimization

Compiler-supported Profiling

Future Work
- Full SMP Support
- Compile BIFs
- Optimize with Knowledge of the Heap

Questions

Future Work: Full SMP Support

Currently:

- Profiling and tracing by one scheduler.
- All schedulers run native code.
- Breakpoints and purge broken.

In the future:

- Cooperative profiling and tracing by all schedulers.
- Full support for purge and breakpoints.

Future Work: Compile BIFs

Currently:

- We only JIT-compile the interpreter loop.
- BIFs are opaque.

In the future:

- Extend JIT-compilation to include BIFs.

Future Work: Optimize with Knowledge of the Heap

- Eliminate the construction of objects on the heap when they are not used:
  `{ok, R} = make_result()`
- Replicate what HiPE does.
- With a JIT-compiler we should be able to do this across modules.
- Attempt to make this generic enough to handle all forms of boxing/unboxing.

Outline

Background

BEAMJIT from 10000m

BEAMJIT-aware Optimization

Compiler-supported Profiling

Future Work

Questions

Questions?