BEAMJIT – A Just-in-Time Compiling Runtime for Erlang

Frej Drejhammar   Lars Rasmusson
Swedish Institute of Computer Science (SICS Swedish ICT)
frej@sics.se, lars.rasmusson@sics.se

Abstract

BEAMJIT is a tracing just-in-time compiling runtime for the Erlang programming language. The core parts of BEAMJIT are synthesized from the C source code of BEAM, the reference Erlang abstract machine. The source code for BEAM's instructions is extracted automatically from BEAM's emulator loop. A tracing version of the abstract machine, as well as a code generator are synthesized. BEAMJIT uses the LLVM toolkit for optimization and native code emission. The automatic synthesis process greatly reduces the amount of manual work required to maintain a just-in-time compiler as it automatically tracks the BEAM system. The performance is evaluated with HiPE's, the Erlang ahead-of-time native compiler, benchmark suite. For most benchmarks BEAMJIT delivers a performance improvement compared to BEAM, although in some cases, with known causes, it fails to deliver a performance boost. BEAMJIT does not yet match the performance of HiPE mainly because it does not yet implement Erlang specific optimizations such as boxing/unboxing elimination and a deep understanding of BIFs. Despite this BEAMJIT, for some benchmarks, reduces the runtime with up to 40%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers, Optimization

Keywords JIT; Erlang; LLVM; Clang

1. Introduction

The Erlang programming language compiles (in the reference implementation) to an abstract machine called BEAM. This paper describes the implementation of a just-in-time compiling runtime for Erlang called BEAMJIT in which the central parts of the just-in-time compiler are synthesized from the C-source code of the BEAM implementation. The LLVM compiler construction toolkit is used for optimization and emission of native code, to reduce the implementation effort.

BEAMJIT uses a tracing strategy for deciding which code sequences to compile to native code. It achieves an up to 40% reduction in benchmark runtime for benchmarks where the tracing strategy selects a suitable code sequence for compilation to native code. Analyzing benchmarks which do show degraded or unchanged performance shows that this is due to a failure of the tracing strategy, the current lack of Erlang-specific optimizations or excessive compilation overhead rather than limitations in the synthesis process.

Although this paper describes the analysis of an abstract machine and synthesis of a code generator for BEAM, the methods described make few assumptions on the underlying implementation and can be applied to almost any abstract machine implemented in C.

Extending an abstract machine implementation by synthesizing the major parts of a just-in-time compiler from its C implementation has two advantages. Firstly, it becomes sufficient to maintain only one implementation for the instructions, instead of two (one for the emulator and one for the JIT compiler). Secondly, the JIT compiled code will conform to the reference implementation even if the instruction set is not documented, as is the case for BEAM. Provided that the synthesis tool is correct, a synthesized implementation automatically stays up to date with the BEAM interpreter and also preserves implementation quirks and bugs which deployed software may rely on. During the development of BEAMJIT the implementation has been forward-ported to one major and three minor releases of Erlang with only minor changes to the synthesis framework.

The Erlang reference implementation includes an ahead-of-time compiler called HiPE [23]. BEAMJIT provides a number of improvements not currently provided by HiPE. These include cross-module optimization and target runtime independent BEAM-modules. HiPE includes handcrafted optimizations for certain common instruction sequences.

The emergence of the LLVM compiler construction toolkit [16] and its accompanying front-end for the C-family of languages, Clang, has made it possible to eliminate much of the drudgery associated with implementing a Just-in-time compiler. The LLVM toolkit operates on a language and target independent low-level intermediary representation, called IR. The toolkit contains a large collection of optimizations which operate on the IR as well as code generators which produce native code from the IR. Language specific front ends, such as Clang, produce IR which is then optimized and emitted as target executable code. The libClang library is a C interface to the Clang C compiler which allows programmatic access to the abstract syntax tree (AST) of a parsed C module.

In the BEAMJIT system the C implementation of the BEAM emulator loop is analyzed by a tool called the *Emulator slicer* which builds a database describing the emulator. The *Emulator slicer* uses the libClang library to extract the AST (*Abstract Syntax Tree*) for the BEAM emulator. Using the AST for the BEAM interpreter loop, the source code implementing each instruction is broken up into basic blocks and a control flow graph for each BEAM instruction is created and added to the slicer database. During this process, local variables are identified and their uses and definitions are added to the database. To simplify code generation and mode switching each basic block in the emulator is also extended with liveness information for each local variable.

Erlang '14, September 5, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3038-1/14/09.
http://dx.doi.org/10.1145/2633448.2633450

Code generation uses as input an execution trace of basic block identities and program-counter tuples, together with a collection of pre-compiled IR fragments. The fragments replicate the semantics of each basic block in the execution trace, and are generated by the emulator slicer during the preprocessing stage at system build-time. During preprocessing, the C source code for each basic block is compiled to IR using Clang. During run-time code generation the execution trace and liveness information from the *Emulator slicer* are used to build an IR-function representing the recorded execution trace. Finally the function is optimized and emitted as native code using LLVM.

The next time the interpreter reaches the first instruction in the trace it will execute the native code for the trace instead of interpreting.

In the current implementation of BEAMJIT only code from BEAM's emulator loop is considered for native compilation. It limits the optimizations that can be done by LLVM, such as inlining of BIFs and boxing-unboxing eliding. Implementations of these optimizations are planned for future versions of BEAMJIT.

Contributions

We show that the core parts of a just-in-time compiling runtime can be synthesized from the C implementation of BEAM. The resulting runtime shows promising performance and evident performance problems are not related to the synthesis process.

By using a code transformation tool at system build time to generate the input to the JIT-compiler we avoid having to maintain two, potentially diverging, implementations of the abstract machine's instruction set.

BEAMJIT is able to automatically construct efficient deoptimization code, i.e. code that turns the native code's state into an equivalent state for BEAM. Deoptimization is needed whenever the execution falls off/exits the trace. By analysing the liveness of the variables in the abstract machine implementation at various points in the source code, BEAMJIT creates a number of tailored deoptimization functions for each potential exit point.

Although BEAMJIT targets Erlang's BEAM, the just-in-time compiler synthesizer design and techniques are generic enough to be applied to other abstract machines implemented in C/C++. Likewise the automated extraction and synthesis processes could be used for other purposes than just-in-time compilers, a possible application is for producing an instrumented version of an abstract machine.

Organization of this Paper

This paper is organized as follows: In Section 2 we give an introduction to abstract machine and just-in-time compiler implementation techniques. In Section 3 we give an introduction to the BEAMJIT implementation before going into details in Section 4. The performance of BEAMJIT is evaluated in Section 5 before covering related work and applicability to other languages in Section 6 and Section 7 respectively. The paper concludes with a discussion of the results and future work in Section 8.

2. Background

This section gives an overview and background of abstract machine and just-in-time compiler implementation techniques.

When implementing the inner instruction execution loop of an abstract machine one of three standard techniques tends to be used: a switch statement, directly threaded code or indirectly threaded code. Section 2.1 describes these implementation techniques. Just-in-time compilation is an implementation technique for abstract machines where frequently executed code is, at runtime, compiled to the target platform's native instruction set. Just-in-time compilation systems are further covered in Section 2.2. To decide on which parts of the abstract machine code to compile, a strategy called *tracing* may be used. With tracing, a log is created, containing the exact path taken during execution. The log will contain the most frequently encountered parts that are worth compiling. Tracing is explained in Section 2.3. After having decided on what to compile, optimized native code is produced corresponding to the selected part(s) of the abstract machine program. Section 2.4 gives an overview of the different choices available to the implementer of a native code emitter.

2.1 Abstract Machine Implementation Techniques

Before delving into just-in-time compilation, there are three main techniques to consider for implementing the inner instruction decoding and execution loop in an abstract machine: A switch statement, direct threading and indirect threading. BEAM, by default, uses direct threading but can fall back to a switch-based interpreter if direct threading is not supported by the target compiler. The following sub-sections explain the three different techniques.

2.1.1 Switch Statement

The simplest way to implement an instruction decoding and execution loop in an abstract machine is shown in Figure 1. Here the program counter, `PC`, is simply dereferenced in a switch statement to select and execute the code implementing the instruction.

```
uint_t the_program[] = {0, Imm0, Imm1, 1,...};
Instr* PC = the_program;
while (1) {
    switch(*PC) {
        opcode_0: {
            /* Perform the actions of opcode_0 */
            PC += 3; /* Skip past immediates */
            break;
        opcode_1:
            ...
            break;
    }
}
```

Figure 1: Using a switch-statement for the inner instruction decoding and execution loop.

2.1.2 Direct Threading

A drawback with the method described in Section 2.1.1 is that it is not very fast. Most compilers will translate the switch statement to a range check followed by a jump-table look up. The extra overhead for the range check followed by the look up can be removed by making use of a technique called *direct threading* [5], an example of which is shown in Figure 2.

Direct threading makes use of first-class labels which is a C-extension provided by GCC and other compilers. Opcodes are, in memory, instead of a number represented by the address of the machine code which implements them. The cost for the dispatching to the next instruction is merely an indirect jump.

2.1.3 Indirect Threading

A drawback with both a switch statement, as described in Section 2.1.1, and direct threading, as described in Section 2.1.2, is that they make it hard to efficiently change instruction implementations at runtime. Changing instruction implementations is useful for functionality such as tracing and other forms of instrumentation that need to be enabled and disabled while the program is running.

A technique called *indirect threading* adds an indirection to the instruction dispatching. In the example in Figure 3, we can switch between execution modes by just reassigning the `mode` pointer.

```
uint_t the_program[] = {&&opcode_0, Imm0, Imm1, ...};
Instr* PC = the_program;
goto **PC;
while (1) {
    opcode_0:
       /* Perform the actions of opcode_0 */
       PC += 3; /* Skip past immediates */
       goto **PC;
    opcode_1:
       ...
       goto **PC;
}
```
Figure 2: In direct threading opcodes are replaced with the address of the native code which implements that opcode.

Switching between two execution modes is still fast, but costs one extra memory load. Usually the load can be serviced from the cache.

```
uint_t the_program[] = {0, Imm0, Imm1, ...};
ISet modeA[] = {&&opcode_0_A, &&opcode_1_A, ...};
ISet modeB[] = {&&opcode_0_B, &&opcode_1_B, ...};
ISet* mode = modeA;
Instr* PC = the_program;
goto *mode[*PC];

while (1) {
    opcode_0_A:
       /* Perform the actions of opcode_0 */
       goto *mode[*PC];
    opcode_0_B:
       /* Perform the actions of opcode_0 */
       /* Do something extra for variant B */
       goto *mode[*PC];

    opcode_1_A:
       ...
       goto *mode[*PC];
}
```
Figure 3: Indirect threading: When the dispatch is made via a look up table switching between two execution modes is fast.

2.2 Just-in-time Compilation

Just-in-time compilation is an implementation technique for abstract machines where frequently executed code is compiled to the target platform's native instruction set at runtime.

Turning an abstract program representation into machine code at run-time was mentioned as early as in McCarthy's LISP paper [18]. A historical survey of just-in-time compilation can be found in [2].

Just-in-time compilation is commonly used in languages that can load or create new code at run time. It is also useful for dynamically typed languages, and languages with macros, as a just-in-time compiler can create specialized versions of the program for the types actually encountered during execution.

Popular language implementations for Java, Lua, ECMAScript (JavaScript, ActionScript) today come with built-in just-in-time compilation support. Most common is to compile individual functions as they are encountered. Such per-method compilation is done for Java by HotSpot [15], for JavaScript by JägerMonkey, IonMonkey, V8, etc., by Julia [6] and CLR [19], by Cog for Squeak [21].

2.3 Tracing Just-in-time Compilation

A tracer records partial traces of the program execution, as it goes in and out of functions, modules/libraries and system calls. For maximum efficiency, the aim is to only record the hot path. The trace is optimized by removing redundant checks and branches. The tracing just-in-time compiler turns the trace into native machine code, rather than compiling the individual functions separately. Multiple traces with the same entry point can be compiled together to form a trace tree. The leaves of the trace tree are jumps back to the root of the tree.

Tracing for runtime optimization was first implemented in Dynamo [3]. Some popular implementations with tracing are Java's HotpathVM [10] and Dalvik [8], JavaScript has TraceMonkey[1], ActionScript has Tamarin-tracing [7], Lua has LuaJIT [22], Haskell has Lambdachine [26], and Python has PyPy. A modified HotSpot [13] and a CIL[2] implementation [4] also support tracing.

PyPy [24] differs from tracing byte-code interpreters (HotSpot, CLR, etc.) by tracing programs written in RPython (a restricted subset of Python) rather than tracing the emulator loop. Typically the RPython program being traced is an interpreter for Python.

Different techniques have been used to decide when to start tracing. I.e. RPython programs are explicitly annotated where tracing should start. Interpreters written in RPython are usually annotated to start tracing at the top of the interpreter loop. The tracing variant of HotSpot [13] annotates the byte code by inserting a virtual instruction to trigger tracing before loops. Other tracers measure the frequency of backward jumps towards a program point, and starts tracing when it is beyond some threshold frequency.

2.4 Native Code Generation Back-ends

A just-in-time compiler needs a back-end able to produce machine code for the specific machine it runs on. Maintaining several back-end tends to be costly. Therefore several languages with purely interpreter-based reference implementations have alternative implementations that target a runtime system with just-in-time compilation support. Python, Smalltalk, Java, Clojure, Scala, Ada, Ruby, PHP, C#, Erlang (Erjang [27]), and many more have alternative implementations that run on the JVM, Microsoft CLR, PyPy, etc. This way, the language will get just-in-time compilation support and a machine code back-end for free from the runtime system. There are also several examples of one language implementing another language for running on a third language's VM, to indirectly provide just-in-time compilation support to the implementation.

An intermediate step between implementing a custom back-end and using another language's virtual machine is to use a back-end library for generating machine code. Compiler back-end libraries such as LLVM [16], libgccjit [17] and GNU lightning [11] have made it increasingly easy to add a native code back-end to a language.

Of particular interest in this paper is LLVM [16] which is the back-end selected for BEAMJIT. The LLVM project calls itself a "collection of modular and reusable compiler and toolchain technologies". LLVM provides a low-level assembly-like program representation called LLVM-IR on which the LLVM toolkit operates. LLVM contains state of the art optimization passes operating on LLVM-IR and also native code generation back-ends for many targets.

LLVM-using Julia [6] is a dynamic language with macros and introspection. It uses LLVM to compile functions created at run-time and specialize them with respect to the argument types encountered. Other languages with implementations that use LLVM include Octave, Python Numba, etc. LLVM is not tracing in itself,

[1] Mozilla replaced TraceMonkey by JägerMonkey as default just-in-time compiler because the cost of falling off the trace was too large. https://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/

[2] CIL is the instruction set of Microsoft CLR

but can be used to optimize and compile traces collected by a tracing interpreter.

3. An Overview of BEAMJIT

BEAMJIT adds just-in-time compilation to BEAM by extending the base system with support for profiling, tracing, native code compilation and support for switching between the three corresponding execution modes: profiling, tracing, and native. During execution the runtime system measures the execution intensity (further described in Section 4.1) of special *profile*-instructions inserted into the instruction stream by the compiler. If the execution intensity rises above a threshold, tracing is started (Section 4.4). During tracing the execution path is recorded and when the runtime system determines that a representative trace is found it is compiled to native code (Section 4.5). To reduce latency, native code generation is performed asynchronously in a separate OS-level thread. Such offloading has previously been used in IBM's WebSphere Real Time Java [9].

The BEAM instructions are high-level, causing the implementation code to contain many if-statements. To take maximum advantage of the recorded execution trace it is therefore beneficial to trace at the granularity of basic-blocks in the interpreter's opcode implementation, and not only at the level of individual BEAM-instructions. To support such detailed tracing efficiently, BEAMJIT's build process constructs a functionally equivalent version of the interpreter which includes basic block tracing. Later on, the just-in-time compiler needs the source code of every recorded basic block in order to generate the LLVM-IR for the native code.

The LLVM project includes a C compiler called Clang, available both as a stand-alone binary and as the library libClang. libClang provides access to the abstract syntax tree (*AST*) of a C module. BEAMJIT uses libClang to analyze the C source of the BEAM abstract machine. The analysis includes extracting the interpreter control flow graph (*CFG*), and is further described in Section 4.2.

Four implementation components are generated from the AST: 1) a tracing version of the interpreter; 2) C-source stubs for mode switching between native and other modes, to be inserted into the BEAM interpreter using the preprocessor macros; 3) C-fragments which are compiled to LLVM-IR by Clang and embedded in the runtime system for the code generation; 4) a second version of the interpreter, called a *Cleanup-interpreter*, used to execute until the next instruction boundary.

The need for a special *Cleanup-interpreter* is motivated by performance degradation observed in early experiments when a single interpreter was generated to handle both profiling and tracing. The reason for the degradation in performance was that the compiler was forced to generate code under the very conservative assumption that native code could return to the interpreter at any basic block. This assumption forced the compiler to allocate most temporaries on the stack which was disastrous for performance. With the *Cleanup-interpreter* execution returns to the interpreter only at BEAM instruction boundaries, see Section 4.4.

At runtime the BEAMJIT abstract machine executes in one of the modes shown in Figure 4. From the profiling mode it either starts tracing or executes already generated native code. If tracing or native code execution stops at an instruction boundary profiling is immediately resumed, otherwise the *Cleanup-interpreter* executes until profiling can be resumed, at the start of the next BEAM-instruction. To support efficient mode switching the directly threaded BEAM interpreter is modified to use indirect threading. Indirect threading allows for changing the implementation of opcodes by just changing the base pointer of the indirection table as described in Section 2.1.3.

BEAMJIT uses LLVM for native code generation. During code generation LLVM-IR fragments representing the basic blocks in

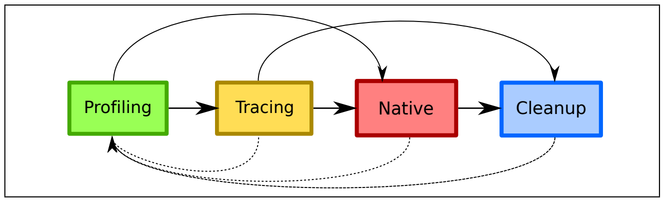


Figure 4: The four execution modes of the BEAMJIT abstract machine.

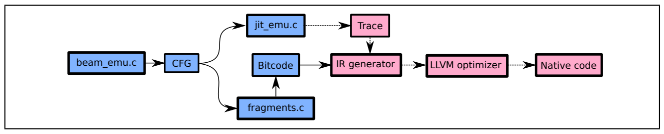


Figure 5: Native code generation in BEAMJIT.

the trace are joined together to form a unique LLVM-IR function for the entire trace. The IR-code is optimized with LLVM's optimization passes, and native code is emitted with LLVM's back-end. Figure 5 shows a schematic view of this process.

4. The BEAMJIT Implementation

BEAMJIT is implemented as extensions and modifications to the reference BEAM implementation. Some extensions are automatically synthesized from the C source code of BEAM, some are implemented by hand. The BEAMJIT implementation also requires support from the Erlang compiler in order to support profiling.

This section describes different aspects of the BEAMJIT implementation. The compiler support for profiling implemented in the Erlang compiler is described in Section 4.1. The preprocessing of the BEAM C source code that is common to all automatically synthesized components is described in Section 4.2. Section 4.3 describes the runtime profiling support. The tracing version of the BEAM interpreter is described in Section 4.4 and native code generation is described in Section 4.5.

4.1 Compiler Supported Profiling

In BEAMJIT the profiler is supported by the compiler. The Erlang compiler inserts BEAMJIT-specific `jit_profile` instructions into the instruction stream to mark locations from where a trace is likely to record a frequently executed instruction sequence. These locations are called *anchors*. The compiler inserts `jit_profile` instructions at the top of loops and at the start of event handling sequences, which in the following discussion are called *streaks*. A *streak* is a a frequently executed trace ending with the process being scheduled out.

Erlang has no explicit looping construct[3] apart from tail-recursion. Therefore the compiler inserts a `jit_profile` instruction at the start of each function. When a tail-recursive function calls itself the anchor will be encountered and the loop detected.

Streaks are a consequence of BEAMJIT's tracer which does not trace across processes and the desire to ensure that event handler loops can be traced and compiled to native code. In Erlang an event handler loop is written as a tail-recursive function which contains a `receive`-statement. If a process is fed a steady stream of messages so that the process never suspends inside `receive`, and control is switched to another process, the tracer will record the loop. If on the other hand the process frequently receives messages, but not often enough to avoid being scheduled out inside `receive`, tracing will never start until the message is handled and the handler

[3] List-comprehensions are compiled to recursive functions

function recursively calls itself. Tracing will then almost immediately be aborted when the process is scheduled out inside the `receive`-statement as no pending messages exist. To enable tracing of *streaks* the compiler also inserts `jit_profile` instructions after `receive`-instructions. The inserted `jit_profile` instruction will guarantee that tracing can start as soon as possible.

4.2 Analyzing the BEAM Source Code

The analysis of the BEAM source code is done in two steps. First the C source code is analyzed to locate and extract the main interpreter loop and to identify individual instructions. This is described in Section 4.2.1. When individual instructions have been identified they are broken up into basic blocks and liveness information is calculated for all variables. This process is described in Section 4.2.2. The tool which does the analysis is called the *Emulator Slicer* and is an Erlang program using a binding to the libClang library.

In this and the following sections we will use examples from a simplified BEAM implementation to illustrate the covered concepts. We focus on a single instruction, `cndbr`, a conditional branch.

4.2.1 Identifying Abstract Machine Instructions

In the source code for the BEAM implementation, the entry point for each instruction is defined by a C preprocessor macro. Figure 6 shows how the `BEGIN_INSTRUCTION` macro is used for the `cndbr`-instruction. The use of C preprocessor macros makes it possible to compile the BEAM either as a switch-based interpreter loop (see Section 2.1.1) or as a directly threaded interpreter (see Section 2.1.2). BEAMJIT leverages these macros for its build process. When the *Emulator Slicer* analyzes the BEAM source code, the entry point macros are overridden to include calls to dummy marker functions. This enables the slicer to unambiguously identify instruction entry points.

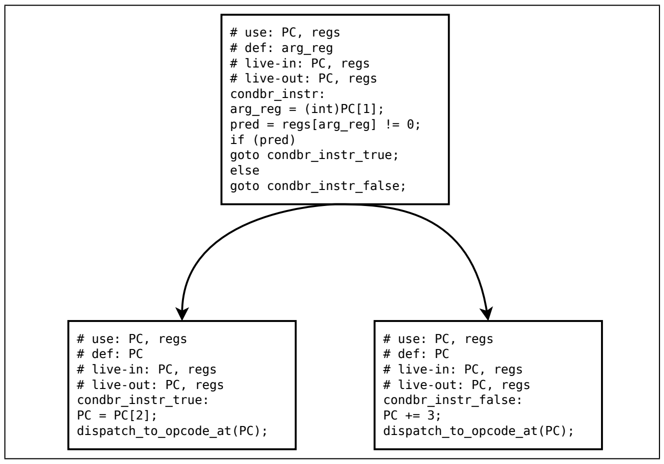
```
#define BEGIN_INSTRUCTION(Name) Name
BEGIN_INSTRUCTION(condbr_instr): {
  int arg_reg = (int)PC[1];

  if (regs[arg_reg] != 0) {
    PC = PC[2]
    dispatch_to_opcode_at(PC);
  }
  PC += 3;
  dispatch_to_opcode_at(PC);
}
```

Figure 6: Pseudo-C for the implementation of a conditional branch instruction: `cndbr <register-index> <immediate-address>`. Control will be transferred to `<immediate-address>` if the register with `<register-index>` is non-zero.

4.2.2 Building the Control Flow Graph (CFG)

When the instruction entry points have been identified, the *Emulator Slicer* traverses the abstract syntax tree for each instruction and builds a Control Flow Graph, *CFG* for each instruction. High-level C control flow constructs, such as `for`, `while` and `do..while`, are converted to simple `if`- and `switch`-statements which branch to explicitly named blocks using `goto`s. A side-effect of building the *CFG* is that local variables which are declared inside C-blocks are renamed and moved to the outermost nesting level. When the *CFG* has been built each basic block is given an unique identity.

When the *CFG* has been built, a liveness analysis on all C-level variables is performed to determine the *use*, *def*, *live-in* and *live-out* sets for each basic block in the implementation of the BEAM instructions. For the running example of the `cndbr`-instruction, it produces the *CFG* shown in Figure 7.



```
# use: PC, regs
# def: arg_reg
# live-in: PC, regs
# live-out: PC, regs
condbr_instr:
arg_reg = (int)PC[1];
pred = regs[arg_reg] != 0;
if (pred)
goto condbr_instr_true;
else
goto condbr_instr_false;
```

```
# use: PC, regs
# def: PC
# live-in: PC, regs
# live-out: PC, regs
condbr_instr_true:
PC = PC[2];
dispatch_to_opcode_at(PC);
```

```
# use: PC, regs
# def: PC
# live-in: PC, regs
# live-out: PC, regs
condbr_instr_false:
PC += 3;
dispatch_to_opcode_at(PC);
```

Figure 7: The portion of the *Emulator Slicer*'s CFG which corresponds to the `cndbr` instruction.

4.3 Runtime Profiling

To identify frequently executed BEAM code to turn into native code, BEAMJIT estimates the execution intensity of each *anchor* using a lightweight profiler. Tracing is started when the execution intensity field exceeds a configurable threshold.

The *anchors* are at locations in the instruction stream identified by the compiler (Section 4.1) designated by `jit_profile`-instructions. The `jit_profile`-instruction contains a pointer to a data structure associated with the *anchor*, and it contains counters used during profiling.

The implementation of the `jit_profile` instruction is written by hand, not by automatic extraction from the BEAM source code. `jit_profile` measures the execution intensity using a global timestamp counter. It is incremented each time an anchor is encountered. The anchor's data structure contains a local timestamp-field recording the global timestamp counter-value of the last time the anchor was visited and an intensity-field. The `jit_profile` instruction checks the timestamp field and only increments the intensity field if the previous timestamp is recent enough, otherwise the intensity is reset to zero. Finally, the anchor's timestamp is set to that of the global timestamp.

4.4 Tracing

The trace is recorded in a trace buffer of fixed size. A trace buffer entry records the basic block identity and the BEAM instruction pointer value, or marks a fork-point (see below).

Tracing starts as a result of the profiler discovering an intensely executed *anchor*. Switching to the tracing version of the interpreter requires the currently live local variables in the profiling interpreter to be transferred to the tracer. This is done by synthesized C fragments which are inserted into the profiling interpreter by redefining the C preprocessor macro (described in Section 4.2.1) which defines the entry point for each BEAM instruction. The synthesized C fragments store all live variables in a structure from which the tracing interpreter initializes its local variables. Tracing continues until one of three things happen: The trace overflows; The process is scheduled out; The trace has not grown during the last *N* iterations, measured by an iteration-field in the *anchor*.

Traces may fork and turn into trace-trees. When, the execution reaches the initial *anchor* tracing continues in a different mode. Instead of adding new entries to the trace, the tracer follows along the already existing entries. If execution diverges from the recorded trace a special trace entry called a *fork-point* is recorded, and normal tracing continues, adding new entries to the trace. When entries are added to the trace, the iteration-field in the *anchor* is cleared. If the trace outgrows the trace buffer, tracing is aborted and the trace buffer is restored to its previous state.

Tracing is accumulative, i.e. a trace can be started and grow until the process is scheduled-out. When the process is scheduled-in tracing continues. Each time the initial *anchor* is visited the iteration-field in the *anchor* is incremented. When the iteration-field rises above a threshold (N) the trace is considered stable and native compilation commences.

Some *anchors* fail to produce good traces, to avoid repeated tracing from a badly chosen *anchor* BEAMJIT implements *anchor* black-listing. In order adapt to changing runtime behavior of the running program, black-listed *anchors* are forgotten after a time. *Anchors* are black-listed when a trace started from the *anchor* repeatedly overflows its trace buffer.

The tracing interpreter itself is synthesized by the *Emulator Slicer* from the *CFG*. It differs from the profiling interpreter in that it only uses `gotos` together with `if` and `switch` for flow-control and all local variables are in a single scope. The parts of the tracing interpreter handling the `cndbr`-instruction is shown in Figure 8. Each basic block of the tracing interpreter starts with a call to a library function which records the basic block identity and the current BEAM instruction pointer. If the library function signals an error, due to an overflowing trace buffer, and the tracing should be stopped, the generated tracing interpreter calls a special *Cleanup-interpreter* which continues execution to the next BEAM instruction boundary.

```
tracing_condbr_instr:
  if (record_trace(PC, tracing_condbr_instr)) {
    cleanup_tracing_instr_condbr(regs, PC,
                                 &live_vars);
    return;
  }
  arg_reg = (int)PC[1];
  pred = regs[arg_reg] != 0;
  if (pred)
    tracing_condbr_instr_true; /* pred == 0 */
  else
    tracing_condbr_instr_false; /* pred == 1 */

tracing_condbr_instr_true:
  if (record_trace(PC, tracing_condbr_instr_true)) {
    cleanup_tracing_condbr_instr_true(regs, PC,
                                      &live_vars);
    return;
  }
  PC = PC[2];
  dispatch_to_opcode_at(PC);

tracing_condbr_instr_false:
  if (record_trace(PC, tracing_condbr_instr_false)) {
    cleanup_tracing_condbr_instr_false(regs, PC,
                                       &live_vars);
    return;
  }
  PC += 3;
  dispatch_to_opcode_at(PC);
```

Figure 8: Pseudo-C for the tracing implementation of the `cndbr` instruction.

The *Cleanup-interpreter* is also generated from the *CFG* . The *Cleanup-interpreter* is structured as a set of tail-recursive functions, one for each basic block, which as arguments take the variables which are *live-in* at that particular basic block. When the function has performed the operations in its basic block, it tail-recursively calls one of its successor blocks according to the flow-control information embedded in the *CFG* until a BEAM instruction boundary is reached. Leaf nodes in the *CFG* which are the last basic blocks in the implementation of a BEAM instruction build a small structure containing the value of the interpreter variables which are live at this point. Before execution is resumed in the profiling interpreter *Emulator Slicer*-generated stubs are used to move the live variables from the structure to the local variables of the interpreter. The *Cleanup-interpreter*-parts resulting from the `cndbr`-instruction are shown in Figure 9.

```
void cleanup_tracing_condbr_instr(
  int regs[],
  Instr *PC,
  live_out_0_t *out)
{
  int arg_reg = (int)PC[1];
  int pred = regs[arg_reg] != 0;
  if (pred) {
    cleanup_tracing_condbr_instr_true(regs,PC,out);
    return;
  } else {
    cleanup_tracing_condbr_instr_false(regs,PC,out);
    return;
  }
}

void cleanup_tracing_condbr_instr_true(
  int regs[],
  Instr *PC,
  live_out_0_t *out)
{
  PC = PC[2];
  out->PC = PC;
  /* regs: The emulator slicer is smart enough to see
     that the pointer to regs never changes */
}

void cleanup_tracing_condbr_instr_false(
  int regs[],
  Instr *PC,
  live_out_0_t *out)
{
  PC += 3;
  out->PC = PC;
}
```

Figure 9: Pseudo-C for the *Cleanup-interpreter* parts for the `cndbr` instruction.

4.5 Native Code Generation

To support runtime native code generation the *Emulator Slicer* generates a C module containing a *CFG* database which can be queried by the runtime code generator. The database stores, for each basic block: the *use* and *def* variable sets; the *live-in* and *live-out* variable sets; the *successor* basic block(s). The database also includes the edges in the *CFG*. Edges are annotated with flow-control information, the information records if the edge represents a conditional or non-conditional branch. For conditional branches the information also includes a pointer to the local variable the `if`- or `switch`-statement operates on and which value the edge corresponds to.

In addition to generating the *CFG* database, the *Emulator Slicer* produces a C module with template functions. The C module is compiled to IR bitcode[4] using Clang. The template function IR is used by the runtime code generator to construct the IR for a recorded trace.

The *Emulator Slicer* creates a template function for each basic block in the *CFG*. The template function is a nullary function which starts with a preamble which declares all variables used by the body of the basic block (given by the union of the use and def sets). The

[4] a binary representation of LLVM-IR

```
int tmpl_tracing_condbr_instr(void)
{
  Instr *PC;
  int *regs;
  int arg_reg = (int)PC[1];
  int pred = regs[arg_reg] != 0;
  return pred;
}

void tmpl_tracing_condbr_instr_true(void)
{
  int *regs[];
  nstr *PC;
  PC = PC[2];
}

void tmpl_tracing_condbr_instr_false(void)
{
  int *regs[];
  nstr *PC;
  PC += 3;
}
```

Figure 10: Pseudo-C for the templates used as input for the Clang-based LLVM-IR generation process for the `cndbr` instruction.

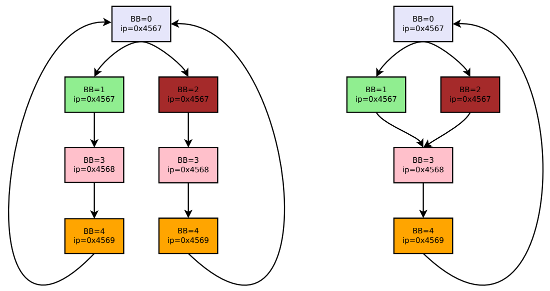


Figure 11: Trace compression identifies and removes duplicated trace segments (left), to produce a simplified trace (right).

template function continues with the body of the corresponding basic block in the interpreter. Basic blocks which do not end in a conditional branch are declared as `void` and return nothing. Basic blocks which end with an `if` or a `switch` return the value of the expression the `if`- or `switch`-statement operates on. Basic blocks which end by dispatching to the next BEAM instruction return the new instruction pointer. The templates for the `cndbr`-instruction are shown in Figure 10.

The runtime system produces traces which represent *loops* or *streaks*. If BEAMJIT has traced both branches of an `if`-statement the trace will contain forking paths. BEAMJIT uses a simple trace representation with very low recording overhead. Because of this, it cannot detect path segments that are shared between branches. To reduce the trace size before code generation, the trace is compressed by identifying and merging common path segments. Compression is performed by building a *CFG* where each node represents a unique trace entry. Figure 11 illustrates this process: On the left we have a trace where following basic block `0`, we can branch to either block `1` or `2`. Regardless of the chosen path, both paths have a common tail of blocks `3` and `4`. By using a hash table to identify identical *CFG* nodes we can reduce the *CFG* to the simpler version on the right.

IR-code generation for a trace starts by creating an IR-level function, the *trace function*, with a signature matching the live-in set of its entry basic block.

The second step is to traverse the trace while consulting the *CFG* database to look up and form the union of all variables in the *use* and *def* sets of the basic blocks in the trace. Variables of matching types are then created as function-local variables, variables which are live-in are set to the value of the corresponding function argument.

The third step is to traverse the trace again and copy the bodies of the template functions corresponding to each basic block into the *trace function* while substituting the template's local variables for the variables in the *trace function*.

The fourth step adds flow-control edges between the copied basic blocks. While adding flow-control edges, the flow-control information in the *CFG* database is used to build *guards* which check to make sure that, at runtime, execution will stay on the recorded path. Where the program may branch to a destination not in the trace, a call to the *Cleanup-interpreter* is inserted. When flow-control edges are added and the *CFG* contains a loop, BEAMJIT also inserts a flag to detect if the native code branches to the *Cleanup-interpreter* without executing at least one iteration through the loop. This flag is used to detect native code which no longer is representative and should be purged. A trace which includes the `cndbr`-instruction is shown in Figure 12 and the resulting LLVM-IR is shown as pseudo-C in Figure 13.

Basic block	PC
....	
....	0xBEEA
condbr_instr	0xBEEF
condbr_condbr_instr_true	0xBEEF
....	0xDEAD
....	

Figure 12: An example part of a recorded trace. The trace shows a taken conditional branch.

```
void trace_fun(
  /* Live-in variables */
  ...,
  live_out_0_t *out)
{
  int arg_reg;
  int pred;

  ...
  /* The body of the preceeding basic block */

  arg_reg = (int)PC[1];
  pred = regs[arg_reg] != 0;
  if (!pred) {
    cleanup_tracing_condbr_instr_false(regs,PC,out);
    return;
  }
  PC = PC[2];

  /* The body of the succeeding basic */
  ...
}
```

Figure 13: Pseudo-C representing the unoptimzed LLVM-IR of the trace in Figure 12.

Optimization of the produced IR-code is performed by LLVM's standard set of optimizations together with one additional BEAM-specific optimization pass. The BEAM-specific optimization, called *load-from-code*, runs in conjunction with constant propagation. It is based on the observation that the BEAM interpreter tends to load data from the BEAM-code area and BEAM-code is constant during execution. If *load-from-code* can detect that a load is performed using a constant pointer into the code area, it can perform the load immediately and replace it with a constant. In many cases *load-from-code* triggers further constant propagation which may open up for new applications of *load-from-code*. The optimization typically eliminates all loads related to the fetching of BEAM-instruction immediate arguments. For the example in Figure 13 the *load-from-code* optimization eliminates all loads from the BEAM-code area as can be seen in Figure 14.

In a dynamically typed language like Erlang, many type tests will translate to *guards*. The optimizer will therefore be able to produce specialized native code for the particular type encountered in the trace.

When the IR-code has been optimized by LLVM's optimizers, it is handed off to LLVM's native code generation back-end, MCJIT. MCJIT emits native code, maps it into the address space of the BEAMJIT process and returns a function pointer to the code. The pointer is stored in the *anchor*.

When the profiling interpreter encounters an *anchor* with a function pointer, it calls the native code through a C fragment generated by the *Emulator Slicer* and inserted into the profiling interpreter using the same mechanism as was used when inserting the fragments for calling the tracer.

```
void trace_fun(
  /* Live-in variables */
  ...,
  live_out_0_t *out)
{
  int pred;

  ...
  /* The body of the preceeding basic block */
  pred = regs[13] != 0;
  if (!pred) {
    cleanup_tracing_condbr_instr_false(regs,PC,out);
    return;
  }
  PC = 0xDEAD;

  /* The body of the succeeding basic block */
  ...
}
```

Figure 14: Pseudo-C representing the optimzed LLVM-IR of the trace in Figure 12.

5. Evaluation

The performance of BEAMJIT was evaluated using the same benchmark suite[5] as traditionally used by HiPE. As BEAMJIT currently does not support BEAM's SMP[6] mode, performance results are normalized against a unicore BEAM runtime[7]. The plain BEAM runtime is chosen as the baseline instead of a HiPE-enabled system as currently HiPE provides, for the selected benchmark suite, such a large performance improvement to plain BEAM (Figure 15) that a comparison to BEAMJIT would be uninteresting. The reason for HiPE's large performance advantage is that it provides many low-level optimizations which are yet to be implemented in BEAMJIT. Examples of optimizations which HiPE has but BEAMJIT currently lacks are, for example: Understanding of BIFs[8] which allows many arithmetic and bit-string operations to be optimized. HiPE can also avoid constructing a term on the heap when a tuple is returned as the result of a called (local or inlined) function, a very common pattern in Erlang programs. Replicating these optimizations within the LLVM framework is planned future work.

We have divided the benchmarks into two groups. In the first group, Figures 16 and 17, the performance is reasonable compared to the baseline as BEAMJIT's execution time is between 60 percent to just over 100 percent of BEAM. In the second group, Figures 18 and 19, the execution time is several times longer than BEAM. We focus the performance analysis on explaining those results.

Figures 16 and 18 show two bars per benchmark. The height of the bars is the elapsed wall clock time for running the benchmark with BEAMJIT divided by the elapsed time of running the same benchmark with BEAM. This makes a bar below 1.0 signify that BEAMJIT is faster than BEAM. The left bar shows the elapsed time for BEAMJIT when just-in-time compilation runs synchronously with the benchmark, i.e. when tracing has found a trace to compile execution stops until native code is available. The right bar shows the time for the asynchronous case when compilation is handed off to a separate thread and the abstract machine continues interpreting until native code is available.

The red hatched area in Figures 16 and 18 shows the elapsed time for running the benchmark in the stable state, i.e. when native code is already available, and the green solid bar shows the aggregated time to compile and time to run (from scratch, with a purged code cache). The black bars are truncated to improve the readability of the graph, as their height would otherwise dwarf the other measurements.

Figure 18 shows the benefit of asynchronous compilation on benchmarks `aprettypr`, `decode`, `fannkuch`, `huff` and `partialsums` where synchronous compilation takes more than ten times as long. Figure 19 shows that most of the time is spent compiling rather than executing. With asynchronous compilation the compilation overhead is masked as the emulator is able to continue making progress in interpreting mode and finish the benchmark in about twice the time compared to BEAM.

Several of the benchmarks are small and not relevant for testing the performance of a just-in-time compiler. For instance `barnes`, `nestedloop` and `w_estone` are so small that the relative time to compile is huge compared to the small time to finish the benchmark. This creates a very tall bar. The benchmarks `heapsort`, `decode` and `barnes` run faster than BEAM in the steady state (the red hatched area in Figure 16). We have chosen not to modify the benchmarks to increase their size as that approach would allow us to mask the true cost of native code generation by making the benchmark arbitrarily large.

An interesting observation can be made on the benchmarks `call_*` and `fun_bm` in Figures 16 and 18. They perform the same number of function calls, one in a loop and in the other the loop is unrolled. For BEAM unrolling speeds up execution because it removes some loop tests. For BEAMJIT unrolling actually slows down execution. This is because the tracer finds a very long trace which is costly to compile, as is seen in figures 17 and 19. Interesting to note is that although Figure 19 shows that we spend most of our time in native code BEAMJIT is still slow, something we attribute to the increased cache footprint of the native code.

Tracing is able to straddle module boundaries. This distinguishes BEAMJIT from HiPE, since the latter operates on individual modules. The benchmarks `call_tail_bm_external` and `call_tail_bm_local` both achieve a significant and equal speedup compared to BEAM, see Figure 16.

6. Related Work

The unique aspect of the BEAMJIT is that it uses the emulator C source code for an abstract machine to extract (using libClang)

[5] https://github.com/cstavr/erllvm-bench

[6] Symmetric MultiProcessor

[7] R16B03-1

[8] Built-In-Function: A function exposed at the Erlang-level but implemented in C and transparently invoked by the abstract machine

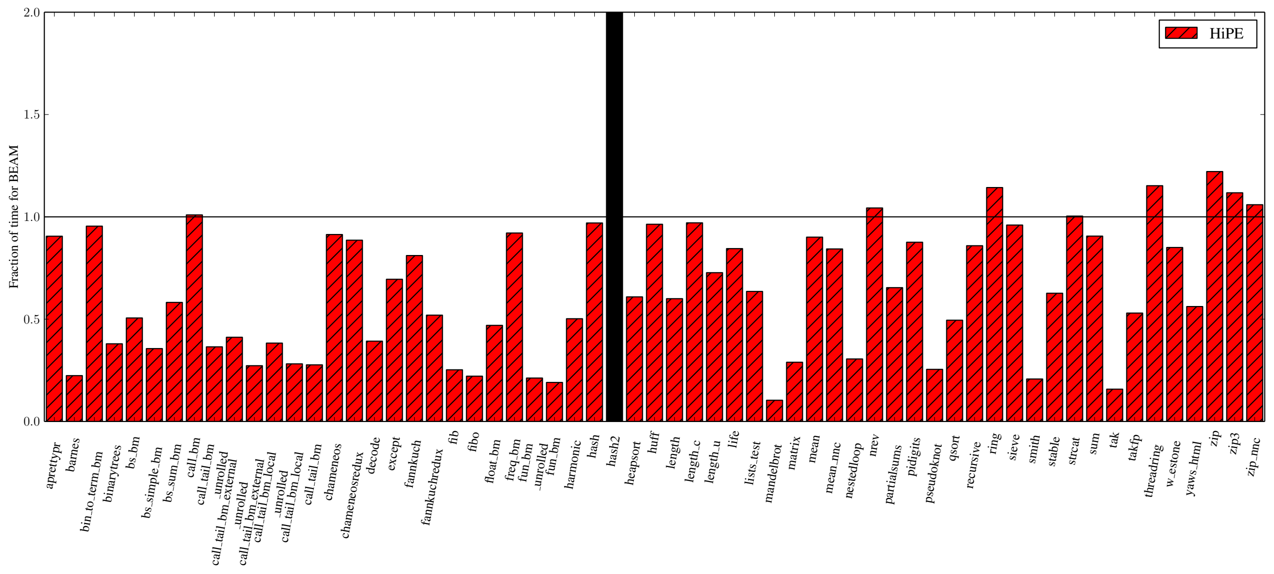


Figure 15: Execution time of HiPE normalized to the execution time of BEAM. Black columns indicates benchmarks for which HiPE is ten, or more, times slower than BEAM.

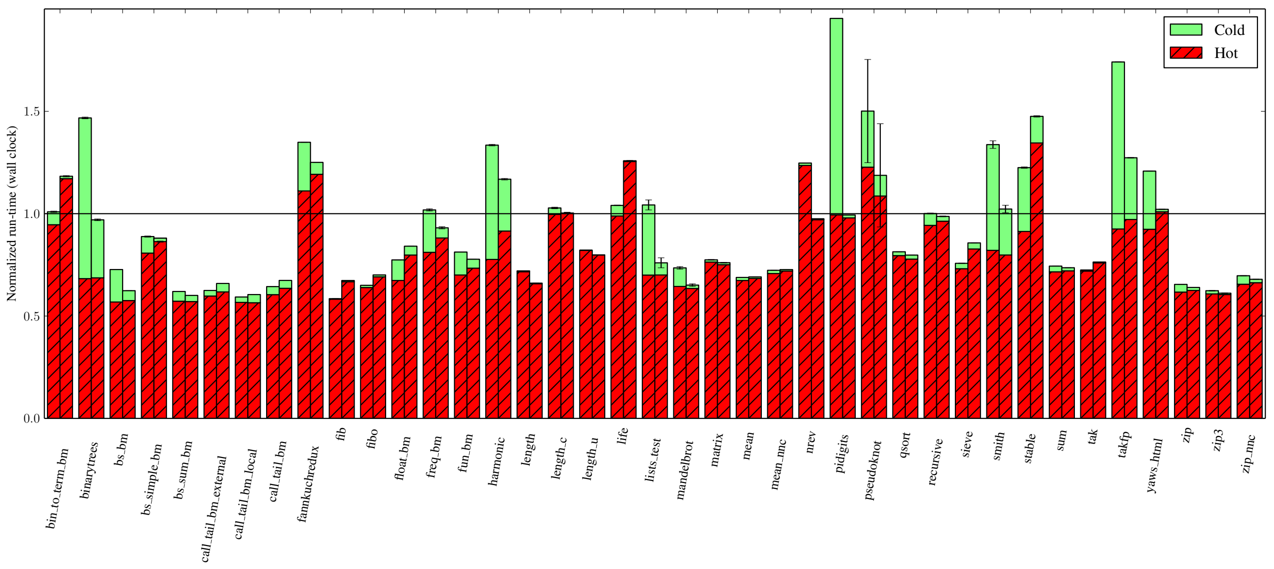


Figure 16: Execution time of BEAMJIT normalized to the execution time of BEAM. The left column is for synchronous compilation, and the right column is for asynchronous compilation. The 1.0-line shows the performance of the unmodified emulator on a single core. The red hatched area (hot) is the elapsed time for running the benchmark in the stable state when native code is already produced. The green solid bar (cold) shows the elapsed time when the benchmark is run with no preexisting native code.

the bulk of the tracing interpreter and the abstract machine instruction implementations. For other parts it borrows heavily from techniques found in other just-in-time compilers (as already covered in Section 2).

BEAMJIT, just as Mozilla's IonMonkey [1], Sun's HotSpot [15] as well as Google's V8 Crankshaft [20] use an abstract intermediate representation for code optimizations before generating machine code. V8 also uses adaptive compilation, meaning that it creates an initial native code version quickly, and only spends more time to optimize the code if it runs often. BEAMJIT does not do adaptive compilation, although it is possible to implement it by generating the first native code with very few LLVM optimization passes active. If the native code later on turns out to be very frequently executed the trace can be recompiled using a higher optimization level.

Two virtual machines and tracing just-in-time compilers for Java are very similar to BEAMJIT: Häubl and Mössenböck's [13] system, henceforth called TraceHotSpot; and Inoue et al. [14]'s system, henceforth called J9Trace. Both TraceHotSpot and J9Trace modifies an already existing Java just-in-time compiler, HotSpot [15]

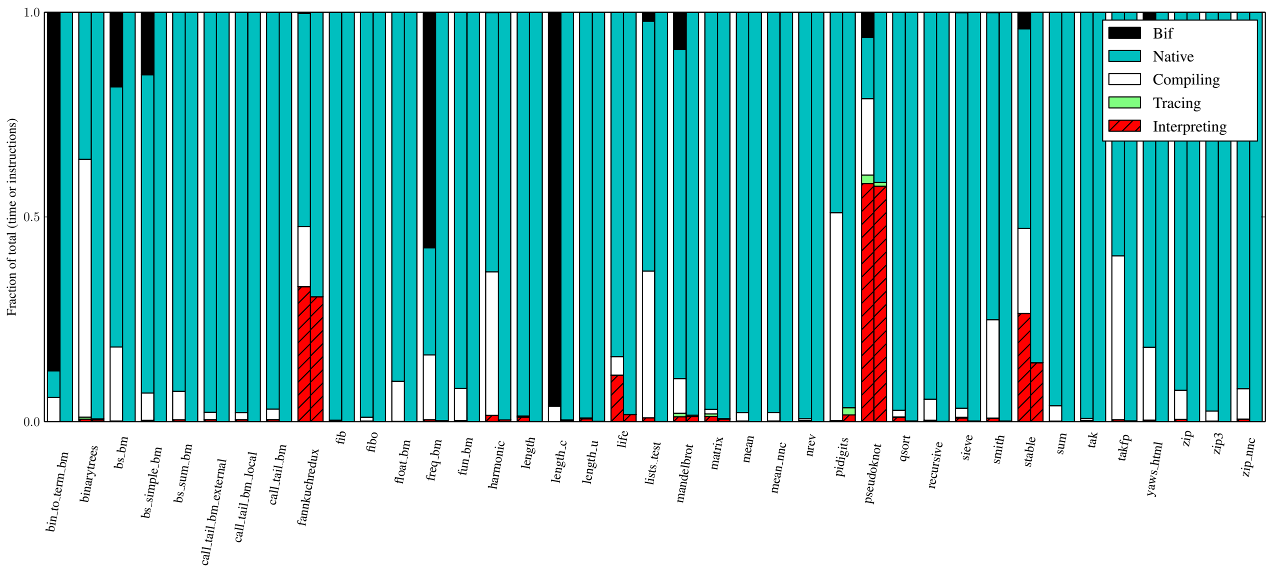


Figure 17: Time spent in different execution modes. For each benchmark, the left column shows the fraction of wall clock time elapsed in each execution mode, and the right column shows the fraction of BEAM instructions executed in the respective mode.

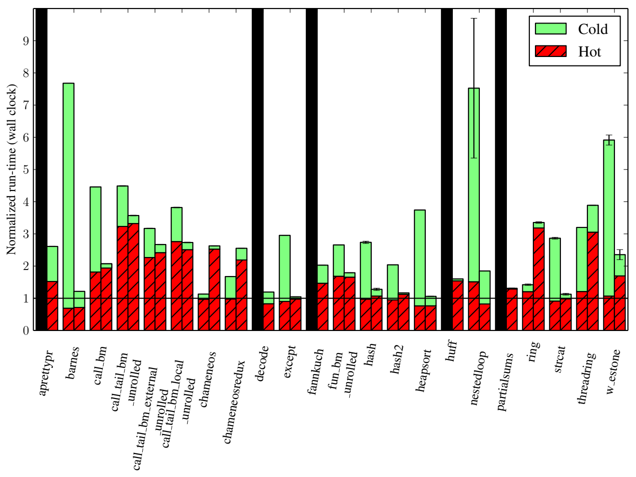


Figure 18: Execution time of BEAMJIT normalized to the execution time of BEAM. The figure shows the benchmarks where BEAMJIT fails to provide a performance improvement. They are discussed in the text. Black columns indicates benchmarks for which BEAMJIT is ten, or more, times slower than BEAM.The red hatched area (hot) is the elapsed time for running the benchmark in the stable state when native code is already produced. The green solid bar (cold) shows the elapsed time when the benchmark is run with no preexisting native code.

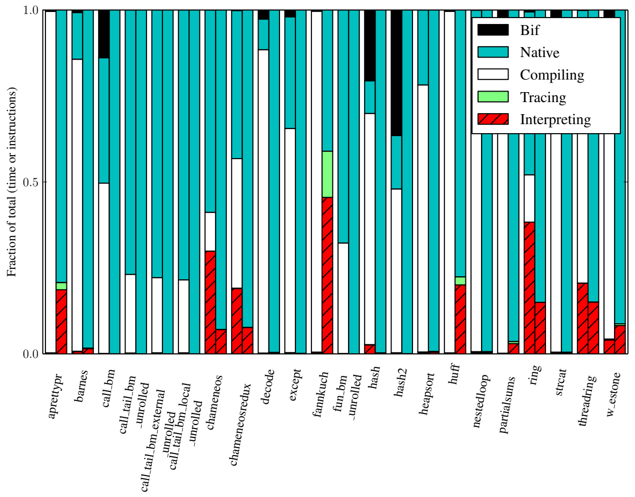


Figure 19: Time spent in different execution modes. For each benchmark, the left column shows the fraction of wall clock time elapsed in each execution mode, and the right column shows the fraction of BEAM instructions executed in the respective mode.

and J9/TR [12] respectively, to be trace based and reuses their respective optimizer and native code emitter. Both system use lightweight intensity based profiling to determine when tracing should start, but in contrast to BEAMJIT they analyze the Java bytecode in a preprocessing step to find anchors.

Of the two, TraceHotSpot uses a trace representation much more advanced than BEAMJIT. TraceHotSpot restricts traces to stay inside methods, when execution transfers to another method, the traces are explicitly linked. The trace linking avoids constructing large traces which must be compiled as a whole and cannot be reused as parts of other traces.

Both TraceHotSpot and J9Trace emit native code which does not maintain the stack of the interpreter. Therefore they both support rebuilding the interpreter stack when execution leaves the fast path. TraceHotSpot uses HotSpot's deoptimization framework to dynamically rebuild the stack. J9Trace uses a native code sequence generated at compilation time to rebuild the stack, a mechanism very similar to BEAMJIT.

ErLLVM [25] is a modified version of HiPE in which HiPE's native code generation back-end is replaced with LLVM. The main motivation for ErLLVM is to gain a state-of-the art native code emitter which provides instruction selection, instruction scheduling and can exploit SMD-extensions while retaining backwards compatibility. As ErLLVM only replaces the final stages of the HiPE compiler it retains all of the HiPE drawbacks compared to a just-in-time compiler.

7. Applicability to Other Language Runtimes

This paper describes a just-in-time compiler for Erlang where the bulk of the tracing interpreter and the abstract machine instruction implementations are extracted from the C source code of the abstract machine, it could equally well be applied to other abstract machine based programming languages where the abstract machine is implemented in C.

The only hard requirement for applying the *Emulator Slicer* is that it must be able to identify the entry point of abstract machine instructions. If a prospective target does not use a preprocessor mechanism to mark abstract instruction (as described in Section 4.2.1), the *Emulator Slicer* could be extended to identify them by, for example, recognizing the form of the C label marking instructions.

Worth noting is also that the *Emulator Slicer*'s analysis of the abstract machine source code can be used for many things apart from a just-in-time compiler. By extending the *Emulator Slicer* it can easily be used for generating instrumented versions of an abstract machine. The way BEAMJIT generates native code using the *Emulator Slicer* and Clang to produce LLVM-IR is not limited to a just-in-time compiler, it could equally well be applied in an ahead-of-time compiler.

8. Conclusion and Future Work

This paper has introduced a just-in-time compiling runtime for Erlang called BEAMJIT. The evaluation shows that a tracing just-in-time compiler for the BEAM abstract machine which is synthesized from the C implementation of the abstract machine gives tangible performance improvements in many cases. Although not all benchmarks in the benchmarking suite show performance improvements, the lack of performance is not due to the synthesis process but rather to correctable inefficiencies in the surrounding runtime environment.

The automatic extraction process greatly reduces the amount of manual work required to maintain a just-in-time compiler as it automatically tracks the base system. During the development of BEAMJIT the implementation has been forward-ported to one major and three minor releases of Erlang with only minor changes required to the synthesis framework.

For the future we have identified a number of areas of improvement. In the runtime system we plan to add full SMP-support, both for executing native code and for performing tracing. SMP-tracing will require a much refined trace representation. We also aim to reduce the overhead of mode switching and also allow more of the BEAM implementation to be just-in-time compiled by extending the tracing process to built-in primitives. Further planned work is to implement the Erlang-specific optimizations mentioned in Section 5 that HiPE has pioneered.

Acknowledgments

This work is funded by Ericsson AB. The authors also wish to thank Roberto Castañeda Lozano and the anonymous reviewers for their helpful comments on this manuscript.

References

[1] D. Anderson. IonMonkey in Firefox 18. https://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/, 2012. Visited May 2014.

[2] J. Aycock. A Brief History of Just-in-time. *ACM Comput. Surv.*, 35 (2):97–113, June 2003. .

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *SIGPLAN Not.*, 35(5):1–12, May 2000. .

[4] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A Trace-based JIT Compiler for CIL. *SIGPLAN Not.*, 45(10):708–725, Oct. 2010. .

[5] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973. .

[6] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. http://arxiv.org/abs/1209.5145, 2012. Visited May 2014.

[7] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 71–80. ACM, 2009. .

[8] B. Cheng and B. Buzbee. A JIT Compiler for Android's Dalvik VM. http://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf, 2010. Visited May 2014.

[9] M. Fulton and M. Stoodley. Compilation techniques for real-time java programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 221–231. IEEE Computer Society, 2007. .

[10] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153. ACM, 2006. .

[11] GNU. GNU lightning 2.0. https://www.gnu.org/software/lightning/, 2013. Visited May 2014.

[12] N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Virtual Machine Research and Technology Symposium*, pages 151–162. USENIX, 2004.

[13] C. Häubl and H. Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 129–138. ACM, 2011. .

[14] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 246–256. IEEE Computer Society, 2011.

[15] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. .

[16] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO 2004. International Symposium on Code Generation and Optimization, 2004*, pages 75–86, March 2004. .

[17] D. Malcolm. Just-In-Time Compilation (libgccjit.so). http://gcc.gnu.org/wiki/JIT, 2013. Visited May 2014.

[18] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, Apr. 1960. .

[19] Microsoft. Common Language Runtime (CLR) 4.5. http://msdn.microsoft.com/en-us/library/k5532s8a#running_code, 2012. Visited May 2014.

[20] K. Millikin and F. Schneider. A New Crankshaft for V8. http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html, 2010. Visited May 2014.

[21] E. Miranda. Build me a JIT as fast as you can... http://www.mirandabanda.org/cogblog/2011/03/01/build-me-a-jit-as-fast-as-you-can/, 2011. Visited May 2014.

[22] M. Pall. LuaJIT 2.0. http://luajit.org/luajit.html, 2014. Visited May 2014.

[23] M. Pettersson, K. F. Sagonas, and E. Johansson. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Proceedings of the 6th International Symposium on Functional and Logic Programming*, FLOPS '02, pages 228–244. Springer-Verlag, 2002.

[24] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953. ACM, 2006. .

[25] K. Sagonas, C. Stavrakakis, and Y. Tsiouris. Erllvm: An llvm backend for erlang. In *Proceedings of the Eleventh ACM SIGPLAN Erlang Workshop*. ACM, 2012.

[26] T. Schilling. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages*. PhD thesis, School of Computing, University of Kent at Canterbury, April 2013.

[27] K. K. Thorup. Erjang: A virtual machine for erlang which runs on java. http://github.com/trifork/erjang/wiki. Visited May 2014.