MASTER'S THESIS | LUND UNIVERSITY 2015

A Tracing JIT Compiler for Erlang using LLVM

Johan Fänge

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-16

A Tracing JIT Compiler for Erlang using LLVM

Johan Fänge
johanfange@gmail.com

June 9, 2015

Master's thesis work carried out at Ericsson Shanghai R&D.

Supervisors: Jörn Janneck, jorn.janneck@cs.lth.se
Haitao Li, lihaitao@gmail.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

We have modified the Erlang runtime to add support for a tracing just-in-time (JIT) compiler, similar to Mozilla's TraceMonkey.

Tracing is a technique to augment an existing interpreter with a JIT simply by recording the instructions executed during a loop iteration, and then generate optimized native code from this. Tracing compilers are particularly suited to optimize number crunching tight loops, an area where Erlang traditionally has been lacking. We make use of the LLVM compiler library to optimize and emit native code.

In micro benchmarks we show some major improvements, reducing execution time by up to 75%.

However, from an engineering point of view, we conclude that the effort of an industrial strength implementation would be substantial – essentially reimplementing large parts of Erlang's interpreter – and discuss a potential solution based on recent research in the area.

Keywords: erlang, tracing, jit, llvm, beam

Acknowledgements

I would like to thank my supervisor Jörn Janneck for encouraging me and putting the foot down when I get too ambitious.

I would like to thank my lovely wife Qian Wu for supporting me while I combined a full-time job with my thesis.

I would especially like to thank my supervisor Haitao at Ericsson who implemented a first prototype, without which I would never have attempted this thesis.

Contents

Chapter 1
Introduction

Erlang [1] is a functional language developed for writing highly concurrent telecom switches with high availability requirements. It has served this purpose well over the years. Erlang has traditionally had a rather high overhead for simple algorithms implemented in pure Erlang, and performance has often been increased by ”throwing more hardware at the problem”.

Nowadays its use has spread out to other areas less interested in the reliability and fault-tolerance, instead focusing on the concurrency. Whereas traditional users of Erlang have been wary of major modifications to the virtual machine (VM), fearing it could become less stable, new users are not as concerned with this, willing to trade performance for reliability. E.g. they may choose to always compile with HiPE [29] (a native code compiler for Erlang) enabled, since the performance increase translates to a reduction in the number of nodes (machines) required. This is especially so for applications running on so called elastic clouds where a reduce in average (rather than only peak) CPU usage translates to savings.

With this in mind we explore a technique called *tracing just-in-time (JIT) compilation*[15, 6, 8, 3], which is particularly suited for increasing the performance of the type of virtual machine used by Erlang. Erlang's VM BEAM [1] uses a byte code interpreter, which is a kind of interpreter that can easily be augmented with a tracing compiler with limited implementation effort, compared to a full-blown traditional JIT compiler. This kind of JIT, also used in e.g. Mozilla's Javascript TraceMonkey [15] JIT, is particularly suited to optimize number crunching tight loops, and works by simply recording the byte code instructions executed during a loop iteration into a *trace*, which is then compiled to native code for fast execution.

We augment Erlangs runtime with a tracing JIT using the LLVM compiler library to optimize and emit native code. In micro benchmarks we show some major improvements, reducing execution time by up to 75%.

Furthermore we compare our implementation to BEAMJIT [9], a project using a similar approach, but with an innovative twist. Whereas we achieve better results on some

benchmarks with our simpler JIT compiler, we conclude that from a software engineering point of view the approach taken in BEAMJIT would be preferable. Furthermore, our results show that BEAMJIT — and also HiPE — still have room for improvements, and that perhaps a hybrid approach could be used, at least temporarily.

1.1 Outline

The rest of this thesis is organized as follows. Chapter 2 provides a thorough introduction to the concepts and tools used in this thesis, along with a cursory overview of some related work. Chapter 3 discusses how we implemented each part of our JIT, touching upon some problems encountered and solutions thereto. Chapter 4 presents a performance evaluation on a half a dozen benchmarks. In chapter 5 we draw our conclusions and discuss areas for further study. In chapter A in the appendix we exhibit a complete, albeit small, trace.

Chapter 2
Background and Related Work

2.1 Interpreters and Virtual Machines

It is common for modern programming languages to use an *interpreter* and a runtime environment to execute computer programs. An interpreter is a program for executing a program in another language, such as Erlang [1] or Java [21]. The interpreter generally executes a program directly, given in some convenient format (such as plain text), rather than compiling the program source to a native executable file, which is then to be executed. This is a flexible technique with many advantages, such as portability, ease of implementation and safety. The main drawback of interpreters is that they tend to lead to major performance degradation compared to the equivalent code compiled to machine code, as explained in [5]. The difference in execution time between interpreted and compiled native code is called the *interpretive overhead*, e.g. by [34]. Common causes of this overhead is parsing and semantic analysis, or for byte code interpreters (see below), instruction decoding and instruction dispatch; see [13] for.

A technique commonly employed to reduce the interpretive overhead is to compile the source code down to *byte code* [13]. Byte code typically consists of a small number of platform independent simple instructions, similar to an assembly language. The byte code can then be executed in an interpreter without having to lose the advantages of platform independence and the dynamicity possible with a runtime with an interpreter. It is also not necessary to implement a full native code compiler, all the way down to platform dependent native code generation[1], reducing the effort needed to implement the interpreter on different platforms. Furthermore, byte code interpreters are a well-studied area of research and there are well-known techniques to increase the performance of such interpreters.

[1] A native code compiler often compiles code in a number of stages, including lexical analysis, parsing, semantic analysis, various code optimizations and finally native code generation, which in itself may consist of the platform specific stages instruction selection and scheduling, register allocation, and sometimes linking to native libraries (often linking is performed in a separate step).

2.1.1 Dispatch and Direct Threading

A simple and very portable technique [13] for implementing the dispatch for a byte code interpreter in C is essentially an infinite while loop containing a giant switch statement, where each instruction is a case clause. The byte code program that is executed by the interpreter is a sequence of integer tokens (often enums or similar) along with any arguments the instruction has. The tokens are used as arguments to the switch to select the appropriate case-clause containing the implementation of the instruction. Control flow returns to the top of the switch statement again at the end of each instruction. The state variables of the interpreter such as the registers and the stack pointer can then simply be defined as local variables declared before the loop in the same function.

A more efficient technique, which is also the default one used by BEAM, is called *direct threading* [13] and typically relies on a GCC C extension[2]. (BEAM also has support for a switch statement interpreter for compatibility.) When loading the program into memory from disk, the integer tokens in the byte code are replaced with the memory address containing the actual implementation of the byte code instruction. Thus instead of using integers in a switch statement, the memory address in the loaded byte code may be used directly in an indirect jump. Consequently, at the end of each instruction the address of the next instruction is read from the loaded byte code and a jump is performed to this address. A less obvious benefit of this is that when there is an indirect jump at the end of each instruction, rather than a single indirect branch instruction at the beginning of the loop as switch statements often results in, the branch predictor in modern CPUs can make predictions conditional on which instruction is executing, resulting in higher branch prediction accuracy.

2.1.2 Context Threading and Inline Threading

Another dispatch method is *context threading* [4] (also known as call threading and subroutine threading), that generates specialized native code for dispatch so that most indirect jumps are removed (replaced by calls) and those that remain would be handled as well as an indirect jump in a native program by the branch predictor. The idea is to put each instruction into a function, and then during byte code loading straight native code is generated so that for each byte code instruction there is a native call instruction generated to call the corresponding instruction function. Control flow (non-straight code) continues to be handled via indirect jumps. To reduce overhead control flow instructions may be inlined, and to further reduce overhead small instructions may also be inlined.

If one then continues to inline every single instruction, the result is very similar to *inline threading*, an older technique for optimizing direct threaded code described in 1998 for Objective Caml [28] and used in SableVM for Java [14]. Here the idea is to identify basic blocks of instructions, and in essence dynamically create so called *super instructions* by sequentially copying the code for the implementation of each instruction into a buffer, and then call these new instructions instead. This can be said to be a very simple JIT

[2]The widely used GCC C compiler has a number of non-standard extensions to the C language, providing features not covered by standard C (see C99 [33] and C11 [35]). The extension referred to here is called ”Labels as Values” [32], allowing the address of a label to stored and manipulated, and then used in a so called computed goto, typically implemented as an indirect jump.

compiler (see below).

2.2 Just-In-Time Compilation

While modern interpreters have become very efficient, there is still a gap between interpreted and heavily optimized native code. However, it may be very difficult to compile a language as dynamic as Javascript or Ruby efficiently to native code with a traditional ahead-of-time (AOT) compiler. Consider for example a tight loop in Javascript calling Math.min, which is compiled to native code with the function call being inlined for efficiency, or maybe the call is even optimized out if the arguments are constant. Then another piece of code is loaded and executed, perhaps via eval, that redefines the Math.min function to do something else. Suddenly the earlier generated native code is no longer valid, and cannot be used. Since essentially any function can be redefined like this, it would seem that no function can be safely inlined.

A solution then is to use *just-in-time (JIT) compilation* (also called dynamic translation) instead AOT compilation. A *JIT compiler* generates optimized native code just like a traditional AOT compiler, but it delays this code generation until the program is actually executed. Then when new code is loaded or executed that invalidates old assumptions, the generated native code can be scrapped and then recompiled to match the changed environment, or the interpreter can be used instead. This process is called deoptimization, and is also useful for debugging. Delaying code generation until runtime also provides the opportunity to make use of information only available at runtime. For example in Java the HotSpot JIT compiler [26] optimizes dynamic dispatch to a simple function call when there is only one class loaded that implements an interface. A JIT compiler may also generate code taking advantage of all capabilities of the hardware system on which it executes, since backwards-compatibility is not a concern. Furthermore, JITs are well-suited to use so called profile-guided optimization by collecting statistics during program execution to to adapt and specialize the compiled code based on actual input data and observed runtime behavior.

An obvious drawback of compiling during program execution is that compilation may require a substantial amount of time. In fact, a short-running program may be slower when executed with a JIT than when executed in a pure interpreter, due to the time the JIT spends compiling. Thus, an important part of a JIT is to identify parts of a program worth optimizing, mirroring the software engineering rule of no premature optimization. Typical candidates for optimizations are frequently executed methods (or functions) and loops.

2.2.1 Tracing JIT

Traditional JIT compilers such as HotSpot for Java typically compile whole methods. The approach essentially requires writing an AOT compiler for the language, a compiler which also makes use of the extra information available at runtime. Implementing a full compiler for a dynamic language can be challenging and require a substantial effort. This has been referred to as a ”big bang” development effort [36].

Tracing JITs provide a simple alternative to this for languages that already have a byte code interpreter or similar. Notable usages of the technique include Mozilla’s TraceMon-

key [15] for Javascript, LuaJIT 2.0 by Mike Pall[27] and recent versions of PyPy [6] for Python and other languages. An early attempt was the Dynamo project at HP [3], which does not build on a byte code interpreter but rather works on the machine code level directly. The idea of a tracing JIT is to execute a program in the interpreter and record frequently executed sequences of byte code instructions, called *traces*, and then compile these to native code. This way, if a traced loop contains a function call then the instructions of the called function would be recorded just the same, effectively inlining the function.

For example, the body of a tight loop might constitute a trace. Such a trace can be seen as a straight or linear piece of code with no branches (a basic block), which jumps back to the beginning, essentially an infinite loop. This loop (hopefully) covers the most common path taken through the loop. Then for exit conditions or other places where the control flow of the program could differ from the recorded trace so called *guards*, a kind of assertion, are inserted. If the condition of a guard is not met control flow must leave the trace, and return to the interpreter. This typically requires some overhead for a context switch, copying values back to the state variables of the interpreter (such as registers). The same context switch also applies in reverse when entering native code execution from the interpreter.

Thus, it isn't necessary to cover all features of a particular method, only those on the hot path are necessary, the others can be handled by the interpreter. Consequently, a common characteristic of tracing JITs is a somewhat uneven execution speed. When running a successfully traced piece of code, it is blazingly fast, but if for some reason tracing fails, it instead runs at interpreter speed. The worst case scenario is that it's difficult to find a single or a small number of traces that dominates runtime, and instead much time is spent on tracing and recording traces that then aren't used, and total execution time may actually be slower with the JIT. Furthermore, a small change in code, for example a change resulting in the code using a not-yet-supported operation, might dramatically affect the runtime, making it more unpredictable.

As described so far, a trace only represents a single code path, so that if there are two hot paths in a loop only one will be recorded. Thus, a penalty is paid every time execution switches over to the other hot path, and this overhead can greatly impact runtime. A solution to this problem are so called *trace trees* [16], where frequent side exits serve as the starting point of new traces, avoiding the need for context switch. A problem then is that tails may be duplicated, and thus trace trees can be further refined by merging nodes to avoid duplication [8].

2.3 Erlang

Erlang [1, 12] is a functional language with dynamic typing focused on concurrency and fault-tolerance. It features single assignment and advanced pattern matching. Loops are implemented via recursion. Tuples and linked lists (cons cell) are important data structures and strings are simply lists of integers. Atoms are a kind of interned string, not entirely unlike enums in other languages. Integers are not limited in size, and implementations are thus required to promote "small" integers to bignum integers when necessary, rather than overflow or wrap around.

Open Telecom Platform (OTP) is the standard open source distribution of Erlang, called Erlang/OTP together. It comes with a collection of standard libraries. OTP pro-

Listing 2.1: `lists:member` implementation

```
member(_, []) -> false;
member(X, [X|_]) -> true;
member(X, [_|Xs]) -> member(X, Xs).
```

Listing 2.2: `lists:member` BEAM generic instructions, generated via `erlc -S member.erl`

```
{function, member, 2, 2}.
  {label,1}.
    {func_info,{atom,member},{atom,member},2}.
  {label,2}.
    {test,is_nonempty_list,{f,4},[{x,1}]}.
    {get_list,{x,1},{x,2},{x,3}}.
    {test,is_eq_exact,{f,3},[{x,2},{x,0}]}.
    {move,{atom,true},{x,0}}.
    return.
  {label,3}.
    {move,{x,3},{x,1}}.
    {call_only,2,{f,2}}.
  {label,4}.
    {test,is_nil,{f,1},[{x,1}]}.
    {move,{atom,false},{x,0}}.
    return.
```

vides support for building large concurrent and fault-tolerant applications using Erlang.

2.3.1 BEAM

In the standard Erlang/OTP distribution, an Erlang `.erl` source file is compiled and optimized into a machine independent `.beam` file containing byte code instructions that are executed by a virtual machine called BEAM.

BEAM is a register based virtual machine with an interpreter [11]. When a program is executed the instructions stored in a `.beam` file are ”loaded” before execution. This is in fact another simple compilation phase that further specializes the byte code into another set of instructions, so that a single ”generic” instruction stored in the beam file can be converted into multiple ”specific” instructions, which are the instructions that are actually seen and executed by the interpreter. E.g there might be one instruction for writing to register 0, one for writing to another register, one for writing to the stack etc.

Furthermore, in this loading step some very frequent instruction sequences are coded as one instruction [18]. These so called *super instructions* (see also section 2.1.2) can reduce the code size of a loaded interpreted program and reduce interpretation overhead such as instruction decoding. It can also increase indirect branch prediction accuracy by splitting the single indirect branch of an instruction into one branch per new super instruction created from the instruction. This gives the branch predictor in modern CPUs more indirect jumps to work with, reducing the noise and allowing it to make more specific predictions, resulting in higher accuracy as seen in [7].

Listing 2.3: `lists:member` BEAM specific instructions (loaded), dumped from memory by calling `erts_debug:df(member)` from the Erlang console, with some extra annotations for clarity

```
00007F2A42F06BB0: >i_func_info_IaaI 0 member member 2  // member_2.1
              +5:  is_nonempty_list_fx f(member_2.3) x(1)
              +8:  get_list_xxx x(1) x(2) x(3)
             +10:  i_fetch_xr x(2) x(0)
             +12:  i_is_eq_exact_f f(member_2.2)
             +14:  move_return_cr true x(0)
             +16:  i_jit_counter_a 0
             +18: >move_xx x(3) x(1)  // member_2.2
             +20:  i_call_only_f member:member/2
             +22: >is_nil_fx f(member_2.1) x(1)  // member_2.3
             +25:  move_return_cr false x(0)
             +27:  i_jit_counter_a 0
```

To show more concretely what the various steps of an Erlang program looks like on its way to interpretation we here showcase an implementation of the `lists:member` function from the Erlang standard library. `lists:member` searches through a list for a specified item, and returns `true` if a list contains the given item, otherwise returning `false`.

In listing 2.1 we can see the Erlang source code for this function. Making use of pattern matching the definition is concise. The first function clause matches an empty list. The second clause matches the case when the given `X` is also the first element of the list. Finally the last clause is matched if none of the other two matches, and keeps looking for the element further down the list by calling the function itself recursively.

Next, in listing 2.2, the function has been compiled to generic BEAM bytecode instructions, and we can see that this code reads more like imperative code or assembly code. The `test` instruction jumps to `{f,4}` if the test fails, and otherwise continues with the next instruction. `{x,0}`, `{x,1}`, etc. refer to registers in the BEAM VM. `call_only` performs a tail-recursive call to a function.

Finally, in listing 2.3, the generic instructions have been loaded into memory and converted to the specific instructions executed in the BEAM VM. For example the `move` and `return` instructions have been replaced by the `move_return_cr` instruction. Here the two characters `cr` at the end denotes that this is an instruction that takes a constant and register 0 as arguments, respectively. (Other registers are denoted by `x`.) The `i_jit_counter_a` instruction is inserted by our JIT during loading in order to be able to trace stack-unwinding loops, and is not utilized for tail recursive loops.

We end this section by giving an example of a trace. For this function, our JIT records a trace for the loop consisting of inspecting a list element, checking that it's not the empty list, seeing that it's not the list element we are looking for and then moving on to the next list element. The trace consists of the specific instructions `is_nonempty_list_fx`, `get_list_xxx`, `i_fetch_xr`, `i_is_eq_exact_f` (which fails and thus jumps), `move_xx` and finally `i_call_only_f` in order, before jumping back to the start. A version of this trace compiled to LLVM IR (see below) can be seen in listing A.1 in the

appendix.

2.4 LLVM

LLVM [19] is a modular collections of libraries for writing compilers. It is an open source project written in C++ used as the back end of many language implementations, some notable being C, C++ and Objective C (via Clang), D (via LDC), Haskell, Julia and Rust. It is also used in the Mac OS X OpenGL pipeline to compile code specialized for the actual GPU present on the system, emulating missing hardware features on the CPU. LLVM consists of an *intermediate language (LLVM IR)*, an optimizer, and a code generator. It can both generate object code ready to be linked and emit native code on-the-fly using the LLVM JIT. We make use the latter for our Erlang JIT.

LLVM IR is (reasonably) platform independent, and is quite similar to common assembly languages, except that it's in Static Single Assignment (SSA) form, it is typed (with integers of different sizes, structs, pointers, etc.) and has an infinite number of (virtual) registers. LLVM IR can exist in three forms: a human readable textual representation, an on-disk bitcode format suitable for quick loading, and an in-memory representation as C++ objects in an object graph.

The optimizer consists of a collection of around one hundred optimization passes operating on LLVM IR, each pass analyzing, modifying and/or transforming the IR before handling it over to the next pass. Most common optimization techniques are available as passes, such as Constant Folding and Propagation, Common Subexpression Elimination, Dead Code Elimination, Global Value Numbering, Loop-invariant Code Motion, and many more.

2.4.1 Alias Analysis

Some optimization passes rely on alias analysis information. Alias analysis answers the question of whether two pointers refer to the same memory address, i.e. of whether or not they alias. If it can be determined that two pointers never alias then stores and loads to these addresses may be safely reordered (relative to each other). If it can be determined that they always alias, then one can be replaced with the other. If none of this can be determined the optimizer must assume that they may in fact sometimes alias. This means that the optimizer has to be very careful and make very conservative assumptions since e.g. reordering or omitting stores and loads could alter the semantics of the program.

In LLVM, alias analysis is supported in the form of information collecting optimization passes that can be scheduled to run before other passes [23]. The pass that follows it can then query the alias analysis C++ object to determine the alias status of two pointers in the analyzed function. LLVM comes with multiple alias analysis passes and it is fairly straightforward to implement your own, although the architecture is somewhat limited as of today (e.g. there is no way for a pass to indicate that it preserves alias analysis information).

2.4.2 Intrinsic Functions

One light-weight extension mechanism of LLVM is via so called *intrinsic functions* [22], which can be used instead of adding a new LLVM IR instruction. These are just function calls to external functions with special names (start with ”llvm.”) that don’t actually exist, Existing unrelated optimization passes need therefore not be updated to add support, as they must already know how to handle function calls.

Intrinsic functions can e.g. be used for well-known functions such as llvm.memcpy and llvm.sqrt to provide inlined efficient implementations and to allow optimization passes to reason about these functions. They can also be used to mark a value or location in the code for some optimization pass or for the code generator. An example of the latter is llvm.expect which is used to specify a likely value of an expression, which can be used by optimizers and code generators to generate code better tailored for the most likely scenario. Another example is llvm.gcroot which marks values to be considered as roots in a garbage collection algorithm and is part of LLVM’s framework for garbage collection support.

2.5 List Unrolling

On modern computer architectures the speed of memory access has not kept up with the speed of CPUs, which has led to ever deeper caches [10]. The cost of cache misses increases exponentially for each cache level, meaning that an optimized working set size and memory access ordering can increase the performance of an algorithm by multiple orders of magnitude.

List unrolling is an interesting technique to reduce the memory size of functional lists such as those found in Erlang. It is an area of research with a long history that might be worth revisiting for Erlang and the BEAM VM. In *cdr-coding* [20] from the 70s a cons cell (list node) can be merged with the next cons cell, so that instead of taking up two words each, the first one only takes up one word. To achieve this, rather than storing a pointer to the next cons cell in the cdr (the next-pointer), instead the next cons cell is stored directly in this memory location, essentially letting the data structures overlap, and some bit flags in the car (the data field) are used to indicate how the memory location of the cdr should be interpreted. Repeating this process can reduce the size of the list by up to 50%. Note that although cdr-coding requires copy-on-write semantics to support mutation, this is not an issue in Erlang since the lists are immutable.

Erlang has long had problems with the unusually large memory consumption of its default string representation - a linked list of integers. The size of an Erlang string containing ASCII text is 8 times of that of the corresponding string encoded with UTF8, and on the 64bit VM it is 16 times the size. The recommended solution to this problem for text processing code is to use *bit strings* [17] instead, which are a special kind of the Erlang built-in type binary. Erlang binaries are basically chunks of memory, with rich support for pattern matching and the equivalence of list comprehension (bit string comprehension).

There has also been other efforts at addressing the issue of reducing memory usage. When switching from the 32 to the 64 bit emulator, it wasn’t only the size of strings that doubled, rather it was the size of a basic Erlang value that doubled, so the size of the entire heap and the stack also almost doubled. A quick solution (”a hack”) to this was the half-word emulator [25], which reduces the size of Erlang words from 64 to 32 bits again, at

the cost of limiting the size of the heap and stack to 4GB.

2.6 BEAMJIT

BEAMJIT [9] is a project with both similar goals and methods to that of this thesis. It is also a tracing JIT for Erlang using LLVM, but with an interesting twist. BEAMJIT does not duplicate the implementation of the instructions of the virtual machine in its JIT, instead it compiles the original BEAM interpreter using Clang, and extracts all relevant basic blocks. These are then used to stitch together one interpreter for profiling and one for recording. Finally the traces themselves are also assembled from the extracted basic blocks.

So in essence, whereas we trace instruction sequences, BEAMJIT traces basic block sequences. Although there is no major theoretical difference between these two approaches, we find the approach taken by BEAMJIT to be very interesting from a software engineering point of view. By compiling the implementation of the JIT from the interpreter source code, the amount of work needed to update the JIT when the interpreter is updated to a new version is eliminated, or at least greatly reduced. Whether or not this turns out to be the case in practice is however difficult to say. Either way, BEAMJIT claims to have successfully performed at least one such update, whereas we have not, due to the amount of work necessary.

It seems clear from their paper, however, that the initial effort of developing such a system is much greater than the comparable effort for us to implement our JIT. It could however be possible that many of the components of their JIT is reusable with another language and/or another interpreter, which is not the case for us.

2.7 HiPE and ErLLVM

HiPE [29] is a native compiler for Erlang which performs static code analysis and some simple type inference, and has been part of Erlang for over a decade now. HiPE implements the whole pipeline down to providing a custom assembler and native code generation for multiple platforms. Whereas HiPE is considered stable and can provide great speedups it lacks many modern optimization techniques available in modern compilers. Implementing these techniques in HiPE would require a non-trivial effort, and the same applies for implementing back ends for new processor architectures.

An attempt to remedy the shortcomings with the current HiPE architecture is ErLLVM [31], which instead of the custom code generator of HiPE uses LLVM as a backend. Success for ErLLVM has been somewhat limited so far, with little to no speedup, despite the many number of optimization passes available. It does however, mostly match the performance of traditional HiPE. We believe it's a promising path to take for Erlang, both when some of the known rough edges of ErLLVM has been worked out so that it may benefit more from the available optimizations and also, once again, from a software engineering point of view.

Chapter 3
Implementation

The just-in-time compiler was implemented by modifying the canonical Erlang virtual machine BEAM. The virtual machine was modified to support profiling of hot functions, trace recording and execution of compiled traces. Since BEAM is written in C and LLVM is a C++ library our JIT was written entirely in C++, except for the modifications to the runtime system itself. In particular the LLVM C++ API is used to generate LLVM IR which is then optimized and compiled down to native code. See figure 3.1 for an overview.

The build system was modified to add support for C++ compilation for the JIT, linking BEAM using a C++ compiler and to support using LLVM as a library. Various other minor changes were necessary, such as ensuring header files from BEAM containing basic types and important utility functions were usable in C++ as well.

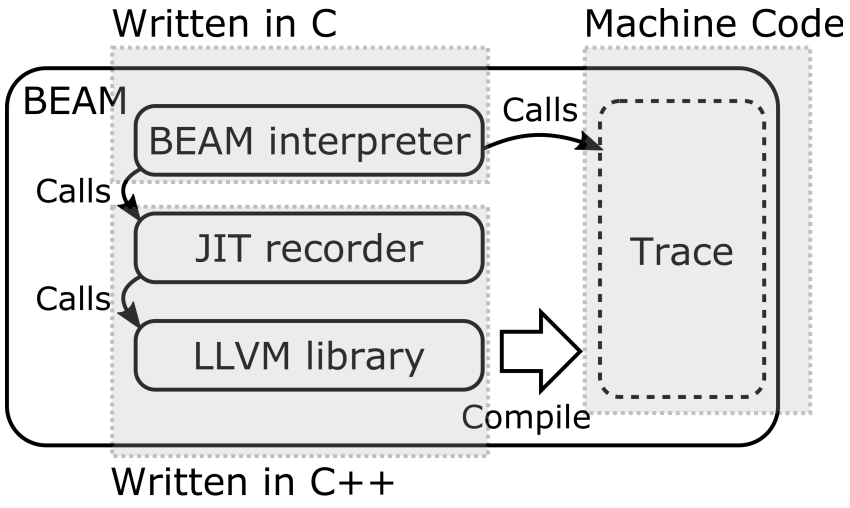

Figure 3.1: An overview of our JIT compiler. The C BEAM interpreter is augmented with a C++ JIT recorder using LLVM, which compiles traces to machine code called from the interpreter.

3.1 Profiling Phase

Usually loop headers such as `while` or `for` loops are chosen as potential trace entry points, but there are no such loop constructs in Erlang, so loops are implemented through tail-recursion. Thus we chose function calls as entry points, and consequently, during profiling we count the number of times a specific function is called. In an attempt to reduce the impact of profiling the counter is stored inline in the function header in the loaded code section. In addition we allow traces to start from return edges in which case we store the counter in a place-holder instruction we insert into the instruction stream (see e.g. listing 2.3). The latter kind is useful for non-tail-recursive functions. Consider a non-tail-recursive sum function that operates on a list. First we trace a stack-building loop consisting of reading a number from the list and placing it on the stack. Eventually we hit the end of the list. Next we trace a stack-unwinding loop that sums the numbers stored on the stack.

Compared to the unmodified virtual machine there is some overhead at each function call (and return). In the current implementation we disable the tracing of functions by default, providing explicit run-time facilities to enable it for a specified function/module, in the form of a built-in function (BIF). Thus, at each function call (and return) we check if this is a function to trace, and if so check the jit counter and typically increase it. If the counter passes a predefined threshold we stop profiling and instead enter the recording phase by starting to record a trace.

3.2 Recording Phase

Recording is implemented by interspersing normal VM instruction execution with recording. That is, our JIT works on a granularity of VM instructions. During recording each instruction is preceded by a call to the recorder written in C++, which directly emits LLVM IR bitcode during recording. To achieve this we have implemented one C++ function per supported instruction of the VM. For unimplemented instructions we have modified a code-generating perl script (which is part of BEAM) to also generate stubs, so that recording is aborted when such an instruction is hit.

3.2.1 Dispatch during Trace Recording

While the default technique BEAM uses for dispatch – direct threading – is efficient, it also means that control flows directly from one instruction to the next via an indirect jump. This makes it difficult to insert a call to the recorder during recording without impacting performance during normal execution. To address this issue we introduced another layer of indirection. This is a fairly common technique which was also used by Mozilla's Tracemonkey (in its interpreter SpiderMonkey) [8, 15] and was described in the BEAMJIT paper [9]. It is also similar to the use of indirection in YETI [36]. At the cost of an extra load, but still without the range check generated for a switch-case interpreter and with an indirect branch at each instruction, this is something in between the two methods provided by BEAM; the loaded program code is represented by opcode tokens, and these tokens are used to index an array containing the memory addresses for the actual instruction implementations.

With this is in place, it is simple to inject recording code into the control flow between instructions. In our implementation there is one array containing the address of each instruction which is used during normal interpretation and another array where all entries redirect to a single recording label. This label calls the recorder and then redirects control flow back to the real instruction implementation using the first array.

3.2.2 Trace Recording and Code Emission

The actual trace recording in our implementation consists of directly emitting LLVM IR bitcode, since this allows access to the complete state of the VM code generation, simplifying implementation. We record each trace into a separate LLVM function, gradually adding code for each instruction as we reach it. Each instruction is given its own basic block, allowing us to easily use an instruction as a jump target, besides also facilitates debugging. After detecting a jump back into the trace completing the loop the trace is complete, and it is optimized and compiled to native code using the LLVM JIT compiler. A production quality JIT may wish to avoid the overhead of emitting code for a trace whose recording is aborted later anyway, and instead just record a light-weight representation until trace compilation. This is less interesting in our case since most aborted traces happen due an unimplemented instruction.

The result is a function pointer that we store in the location originally used for the counter during the profiling phase. Then this function is called via the function pointer the next time the corresponding Erlang function is called (often directly after tracing it if it's a hot loop). The function pointer takes as arguments the various variables of the virtual machine, such as the registers and the program counter. To allow these to be passed back out again these are passed by reference (as pointers). Furthermore, since we prefer the state variables of the interpreter to reside in registers, we copy the variables to and then from temporary variables on the C-stack to avoid pinning the state variables themselves to memory addresses.

We record instructions before they are executed in the VM, since this allows us to analyze the inputs and preconditions of the instruction, and generate more specific and thus faster code, as well as abort recording when we encounter something unsupported or something not worthwhile to record. Erlang's instructions are fairly complex, handling multiple cases such as small ints and bignums in a single instruction. We focus on implementing the simple, fast and optimistic code paths, since these result in good traces. I.e. we focus on the code paths where all integers stay small, all parameters and values are of the expected type, and no errors occur. Thus, e.g. arithmetic instructions begin with guards asserting that we are handling small ints if the input of the instruction being recorded also are small ints.

A number of instructions in BEAM, in particular the super instructions (see section 2.1.2), are implemented using C preprocessor macros in header files generated from a domain-specific language, parsed by a perl program. The resulting header files turned out to be simple enough that it was possible for us to implement most of the instructions by reimplementing the macros to instead call carefully written C++ functions in the recorder. Rather than performing the described operations, these functions then emit corresponding LLVM IR using the LLVM API.

A slight complication of this strategy occurs when the macro instruction implements

some kind of control flow, for example a simple equality check. After the equality check there are two possible paths that may be taken, either to go to the next instruction or to jump to a given label. The inflexibility of the macro solution means that although we can make the macros generate code that emits either code path following the branch, we cannot as easily tell which branch will be taken given the current state of the VM. To solve this problem we delay code generation of the branch until after the branch has executed, that is, right before recording the next instruction.

We use a similar technique to record other control flow such as function calls and function return. This allows us to easily trace across function borders and e.g. inline function calls and fun-calls (known as lambdas in other languages) without closures, simply by recording their destination.

3.3 LLVM Optimization Passes

3.3.1 LLVM Standard Optimizations

We run the optimization passes listed below on our compiled traces. Between each pass we also run the `-instcombine` pass, which is a quick pass that performs various kinds of simplifications. Which passes best to run and the order of them is far from trivial, sometimes lovingly referred to as ”black magic”, and can be seen as a difficult search problem in itself. Running passes takes time, some takes more time than others, and not all passes are appropriate to run after each other. We make no claim as to the appropriateness of the following order, it is simply what we have used so far, and what the results of this thesis are based on. Refer to LLVM's documentation further information [24].

Optimization passes executed, in order:

`-instcombine`: Combine redundant instructions

`-simplifycfg`: Simplify the CFG

`-mem2reg`: Promote Memory to Register

`-reassociate`: Reassociate expressions

guard inlining pass This is our own guard inlining pass. See section 5.2.1 for details.

`-tailcallelim`: Tail Call Elimination

`-gvn`: Global Value Numbering

`-sccp`: Sparse Conditional Constant Propagation

`-dce`: Dead Code Elimination

`-indvars`: Canonicalize Induction Variables

`-licm`: Loop Invariant Code Motion

`-indvars`: Canonicalize Induction Variables

`-loop-unroll`: Unroll loops

3.3.2 Guard Inlining Pass

Lowers guard functions to branches. In section 5.2.1 we describe this in context.

3.3.3 JIT Alias Analysis Pass

We have implemented a special alias analysis pass for JIT traces, which knows about the variables passed in to the function. For example, it knows that the registers and the stack will never alias, which is more than LLVM knows. The pass is based on Scalar Evolution, which we use to compare if two pointers in fact are identical, and if not determine the base pointer of a pointer expression. We have found the execution time to be negligible and consequently we execute it before any pass that might benefit from its information.

3.4 Debugging

Debugging turned out to be quite difficult in practice. Luckily for us we have a real interpreter to compare with to see where things went wrong. To make full use of this we implemented a simple system where we logged various state variables from the interpreter, which we also called from the (debug version) of our JIT. This allowed us to run simple programs both in the interpreter and in the JIT, and then just compare the logs to spot where they diverge. This worked fairly well, except for a difficulty in detecting live registers and variables, which then caused the virtual machine to crash when attempting to print e.g. a dead register containing garbage.

Chapter 4

Performance Evaluation

4.1 `erlang:length` and `lists:member`

We evaluated the performance of our JIT by comparing two built-in functions from the Erlang standard library which are implemented in C, with equivalent implementations written purely in Erlang, and executed with our JIT. The `erlang:length` function simply calculates the length of a list, and thus simply loops through the list until it reaches the end (remember that Erlang uses singly linked lists). Again, the `lists:member` function returns true if a list contains a given item, otherwise returning false, and is the one used as an example in section 2.3.1 above. The Erlang implementations are tail recursive and quite idiomatic. For length we also did some list unrolling. See listing 2.1 on page 13 and listing 4.1 for the implementations.

The result was that compared to the `lists:member` and `erlang:length` reference C implementations, our Erlang implementations were respectively 7,00x and 3,49x slower. With the JIT we obtained a speedup over the Erlang implementations of 6,88x and 3,10x respectively. In other words, our JIT reclaimed most of the lost performance, only resulting in an increase in runtime from the C implementation of 2% and 13% respectively for the `lists:member` and `erlang:length` benchmarks.

Listing 4.1: `erlang:length` implementation

```
length(L) -> length(L,0).

length([_,_,_,_,_,_,_,_|Xs],Acc) -> length(Xs,Acc+8);
length([],Acc) -> Acc;
length([_|Xs],Acc) -> length(Xs,Acc+1).
```

4.2 BEAMJIT Comparison

Due to the many similarities of our JIT with BEAMJIT (see section 2.6), we have attempted to include a small runtime benchmark comparison with BEAMJIT as well. The selected benchmarks are part of the ErLLVM benchmark suite (available on GitHub, see [30]). Our JIT is still unable to trace most of the benchmarks in the suite, and as such we have selected a number of benchmark where our JIT gives a shorter execution time when enabled.

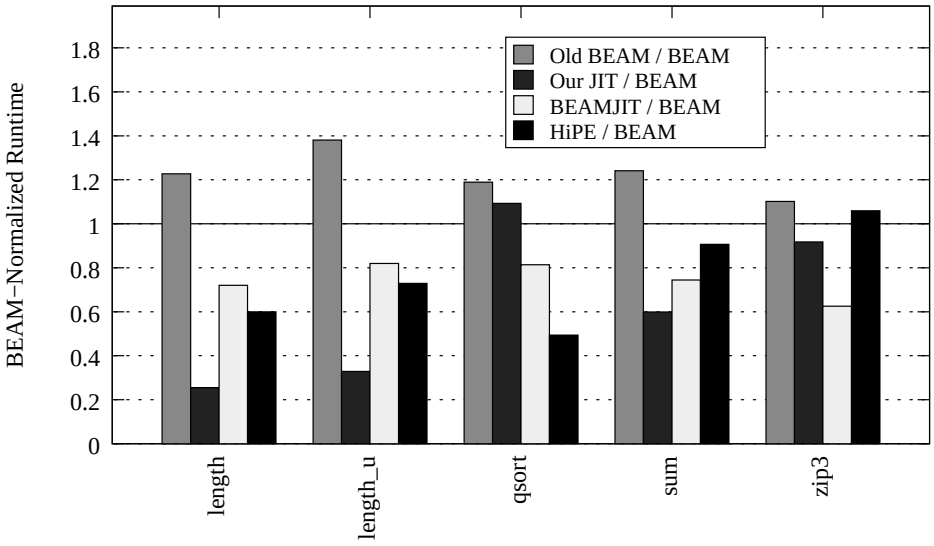

Figure 4.1: The runtime of microbenchmarks from the ErLLVM benchmark suite [30], divided by that of Erlang R16B03-1 BEAM.

A slight complication arises from the fact that the base version of the BEAM VM used by BEAMJIT is newer than the one used in this thesis. In fact, this is indirectly an example of the main benefit of the approach taken by BEAMJIT: facilitating upgrading to a new version. As such, we have two baselines: our version of BEAM without the JIT, denoted ”Old JIT” in figure 4.1, and the newer version of BEAM used in the BEAMJIT paper (see [9]), also without JIT, denoted just ”BEAM”. Both ”Our JIT” and ”Old BEAM” use a modified Erlang R14B02 BEAM. HiPE and BEAMJIT use R16B03-1.

The data for BEAMJIT in the figure comes from the the BEAMJIT paper, and we have normalized all other runtimes to those of the new BEAM in figure 4.1, in order to make them comparable. More specifically we use the total runtime of the non-asynchronous compilation of BEAMJIT, i.e. we include the ”cold” time from the paper, which includes the time taken to record and compile traces.[1]. We do not present any measurement of the fraction of time spent on compiling and tracing in our benchmarks but our preliminary experiments suggest that the distribution would be similar to that presented in the BEAMJIT paper.

The new version of the BEAM interpreter provides a substantial boost over the one we used, as can be seen from the fact that ”Old BEAM” is greater than 1 for all shown the benchmarks. Thus when our JIT fails to trace a benchmark and falls back to the interpreter

[1]That said, this happens to have minimal effect, since the chosen benchmarks spend very little time in compilation.

the expected result is that the runtime is similar to that of ”Old BEAMJIT”. For example, our JIT provides a slight improvement for qsort over ”Old BEAMJIT”, but fails to catch up with the new interpreter, and is far behind BEAMJIT. We expect that upgrading to the latest BEAM version would provide an overall speedup, but have little effect on the benchmarks were our JIT performs well.

Our JIT performs well on simple list-based benchmarks with very tight loops, such as length and length_u, which is quite expected. Interestingly enough we outperform both HiPE and BEAMJIT in these benchmark by a wide margin. BEAMJIT is not able to achieve a 50% runtime reduction on any of the many benchmarks showed in their paper, whereas we achieve 75% reduction for the benchmark ”length”, and hand-optimized trials suggest that there is further room to at least half this runtime by implementing a few simple optimizations (see section 5.2.1).

Chapter 5
Conclusions and Future Work

Here we discuss the results of the evaluation and where to go from here.

5.1 Conclusions

Our JIT manages to provide a substantial performance increase in a few selected benchmarks. However, it is still very limited, and unfortunately still fails to record and compile traces for most benchmarks we have tested. For some benchmarks, e.g. qsort in 4.1, it only manages to record a few traces, and these traces do not cover all hot paths. This situation could be improved using trace trees [16], i.e. start recording traces again on hot side exits. Overall, a full implementation of even a tracing JIT would have required more time than was available for this thesis.

The large difference in the few benchmarks shown in figure 4.1 compared with BEAMJIT could be due to our handwritten recorder emitting higher quality code than the generated counterpart of BEAMJIT, which might include more unnecessary overhead. We suspect one explanation could be that BEAMJIT perhaps fails to properly handle arithmetic functions (which are implemented as so called BIFs in BEAM), which are crucial for these benchmarks, especially addition and subtraction by one. Whatever the reason for the performance difference we clearly show there is room for both BEAMJIT and HiPE to improve.

5.2 Future Work

An interesting approach could be a hybrid approach, where most instructions are traced using BEAMJIT (or similar) and a select few instructions are handoptimized for performance. This could even be implemented as a custom LLVM optimization pass. In general it seems feasible to implement optimization passes to enlighten LLVM about BEAM, possibly with the help of further annotations in the source code, allowing it to generate code

with as high quality as our simple JIT does. Such a hybrid system could automate the bulk generation of code, with some parts still written by hand.

5.2.1 More LLVM Optimization Passes

Here we discuss some further optimizations we have considered but not yet implemented in our JIT.

Use LLVM Allocas for BEAM Registers

Allocas corresponds to allocations on the C-stack in LLVM. Currently, only BEAM's register 0 (which is special cased throughout BEAM) uses a separate alloca in our JIT. All other registers are accessed via a pointer. Implementing this would make them easier to analyze in passes.

Use LLVM Allocas for the BEAM Stack

Currently, the BEAM stack is accessed via a pointer. This makes it difficult to optimize away stack traffic. E.g. calling an Erlang fun (lambda) might require increasing the stack pointer, saving some variable to the newly allocated space, executing the fun, then restoring the variable from the stack and decreasing it again. It is difficult for the LLVM optimization passes to recognize the redundancy as it is implemented now. Delaying writing the stack to memory until looping back or side-exiting through a guard would hopefully make it more transparent.

Lift Allocas to Virtual Registers for the BEAM Stack

Currently, stores and loads from allocas are used for convenience, to avoid the extra work required by the SSA form of LLVM to handle phi-nodes and keeping track of new values for our state variables. This works well, and often but not always LLVM is able to deduce and create the phi-nodes for us. Unfortunately, the optimization pass responsible for this is too conservative, and given our knowledge of the generated code we could implement a simple pass to override this caution where we know it's safe.

Implement Redundant Guard Elimination

We currently implement some guards via a special function. This allows us to easily recognize and identify the guards in the LLVM IR, similar to how LLVM's intrinsic functions work, except that our function actually has an implementation. We have then written a pass that inlines these guards, so that there is no runtime cost. This allows us to detect for example that we already know that a value is a small int, since we have already checked it in an earlier guard. Furthermore, we can detect that certain operations will preserve certain guards. E.g. bitwise OR, AND or XOR between two small ints is guaranteed to produce another small int.

Unbox Small Ints More Explicitly

Currently small ints are unboxed wherever they are needed, with the result boxed again. Such a boxing-unboxing sequence can sometimes be optimized away by LLVM's optimizer, but often not. Perhaps one could store the boxedness of a variable in a separate register, to make it more visible to the optimizer. Or perhaps we could use functions for boxing/unboxing and then implement a pass that specifically looks for these.

Unbox Small Tuples

Currently tuples are not handled efficiently, where handled at all. Instead we could allow each values of a tuple to be handled in a separate register, until control is returned to the interpreter. This applies to small lists too, although this usage is less common. (Lists are typically handled by recursive loops.)

Unbox Records

Erlang records are a implemented as tuples through compile-time macros, with much overhead. These could be implemented as C-structs while inside a trace.

5.2.2 List Memory Optimizations

While experimenting with the benchmarks it became clear that it would be difficult to achieve big improvements with some of them, since they were memory bound, rather than CPU-bound.

While bit strings are efficient in many situations they are not replacements for lists, since they don't support efficient appends. This means that one cannot just silently replace the representation of strings since it could lead to degradations in existing programs assuming the previous efficient behavior. An alternative then could be something like cdr-coding [20], or if random access is also desired the VList [2] could be an alternative. Again, removing the overhead for the links in the linked list is only half the equation for strings. The other half requires choosing an efficient encoding, which often means something like UTF-8 for programming, text-based data formats or western text, and possibly UTF-16 for e.g. Asian texts. Seamless automatic support for strings in Erlang would probably require support for ”packing” arbitrary lists of integers in this manner.

5.2.3 Context-Threading

It would be interesting to implement Context Threading or Inline Threading for BEAM using the LLVM JIT to generate native code. By using LLVM it would also be possible to run some optimization passes on the resulting code during code loading. This would probably work best if based on the technique of compiling BEAM using Clang that is used in BEAMJIT.

Bibliography

[1] Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

[2] Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays. Technical report, Swiss Federal Institute of Technology in Lausanne, LAMP, 2002. (PDF link).

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[4] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proc. of the 3rd Intl. Symp. on Code Generation and Optimization*, pages 15–26, March 2005. (PDF link).

[5] Denise Boiteau and David Stansfield. *Bits and Bytes Program 6, Computer Languages*. TVOntario, 1983. (Online link).

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[7] Kevin Casey, M Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):37, 2007. (PDF link).

[8] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael 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*, pages 71–80. ACM, 2009. (PDF link).

[9] Frej Drejhammar and Lars Rasmusson. BEAMJIT: a just-in-time compiling runtime for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 61–72. ACM, 2014. (PDF link).

[10] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007. (PDF link).

[11] Ericsson AB. *`beam_emu.c` – the implementation of Erlang's BEAM interpreter (emulator)*. `https://github.com/erlang/otp/blob/maint/erts/emulator/beam/beam_emu.c` (Accessed 2015-06-07).

[12] Ericsson AB. *Erlang/OTP documentation*, Erlang/OTP 17 edition. `http://www.erlang.org/doc/` (Accessed 2015-06-07).

[13] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003. (PDF link).

[14] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction*, pages 170–184. Springer, 2003. (PDF link).

[15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

[16] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical report, ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, 2006. (PDF link).

[17] Per Gustafsson. Programming Efficiently with Binaries and Bit Strings. In *Erlang/OTP User Conference*, 2007. `http://www.erlang.org/euc/07/papers/1700Gustafsson.pdf` (Accessed 2015-05-17).

[18] Bogumil Hausman. *The Erlang BEAM Virtual Machine Specification (historical)*. Ericsson Telecom AB, Computer Science Laboratory, 1.2 edition, October 1997. `http://www.cs-lab.org/historical_beam_instruction_set.html#2.15%20Instruction%20Folding` (Accessed 2015-05-28).

[19] Chris Lattner. *The architecture of open source applications: LLVM*, chapter 11. 2014. `http://www.aosabook.org/en/llvm.html` (Accessed 2015-05-28).

[20] K. Li and P. Hudak. A new list compaction method. *Software – Practice and Experience*, 16(2):145–163, February 1986. (PDF link).

[21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[22] LLVM Project. *Extending LLVM: Adding instructions, intrinsics, types, etc.*, LLVM 3.7 edition. `http://llvm.org/docs/ExtendingLLVM.html` (Accessed 2015-05-24).

[23] LLVM Project. *LLVM Alias Analysis Infrastructure*, LLVM 3.7 edition. `http://llvm.org/docs/AliasAnalysis.html` (Accessed 2015-05-24).

[24] LLVM Project. *LLVM's Analysis and Transform Passes*, LLVM 3.7 edition. `http://llvm.org/docs/Passes.html` (Accessed 2015-05-24).

[25] Patrik Nyblom. Presentation: The "halfword" virtual machine. Erlang Factory SF Bay Area conference, 2012. `http://www.erlang-factory.com/upload/presentations/569/Halfword_Erlang_Factory_SF_2012.pdf` (Slides, accessed 2015-05-17) `https://vimeo.com/33544794` (Recording).

[26] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001. (PDF link).

[27] Mike Pall. Luajit 2.0 intellectual property disclosure and research opportunities. *Lua-Users Archive, Nov*, 2:2009–11, 2009.

[28] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300, 1998. (PDF link).

[29] Konstantinos Sagonas, Mikael Pettersson, Richard Carlsson, Per Gustafsson, and Tobias Lindahl. All you wanted to know about the HiPE compiler: (but might have been afraid to ask). In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 36–42. ACM, 2003. (PDF link).

[30] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. ErLLVM Benchmark Suite. `https://github.com/cstavr/erllvm-bench/` (Accessed 2015-04-25).

[31] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. ErLLVM: An LLVM Backend for Erlang. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 21–32. ACM, 2012. (PDF link).

[32] Richard M Stallman et al. *Labels as Values*. Using GCC: the GNU compiler collection reference manual. Gnu Press, 2003. `https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html` (Online Edition, Accessed 2015-06-05).

[33] Guy L Steele and Samuel P Harbison. *C: A Reference Manual*. Prentice Hall PTR, 2002.

[34] Clark Wiedmann. A performance comparison between an APL interpreter and compiler. *ACM SIGAPL APL Quote Quad*, 13(3):211–217, 1983.

[35] Wikipedia. C11 (C standard revision) — Wikipedia, The Free Encyclopedia, 2015. `http://en.wikipedia.org/w/index.php?title=C11_(C_standard_revision)&oldid=664230762` (Accessed 2015-06-05).

[36] Mathew Zaleski, Kevin Stoodley, and Angela Demke Brown. YETI: a graduallY extensible trace interpreter. In *VEE '07: Proceedings of the 3rd ACM/USENIX international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007. ACM Press. `http://www.cs.toronto.edu/syslab/pubs/vee11-zaleski.pdf`.

Appendices

Appendix A
Code Listings

A.1 `lists:member` Trace

Below is the optimized LLVM IR emitted by our JIT for the trace of the hot loop of the erlang:member benchmark. Compared to most benchmarks this is a short trace; most benchmarks have multiple, much longer traces. The branches to side_exit corresponds to guards in the trace, after they have been lowered to simple branches.

Listing A.1: LLVM IR for the `lists:member` trace

```
define void @"member:member@2"(%struct.process* noalias %"process*",
    i64* noalias %"reg*", i64* noalias %x0, i64* noalias %tmp_arg1,
    i64* noalias %tmp_arg2, i64* noalias %i0, i64 %fcalls, i64*
    noalias %htop, i64* noalias %stop) {
entry:
  %0 = load i64* %x0, align 8
  %.phi.trans.insert = getelementptr i64* %"reg*", i64 1
  %.pre = load i64* %.phi.trans.insert, align 8
  %1 = getelementptr i64* %"reg*", i64 3
  %2 = getelementptr i64* %"reg*", i64 2
  %sub_fcalls = add i64 %fcalls, -1
  %struct.process.field.fcalls2 = getelementptr %struct.process*
      %"process*", %i64 0, i32 14
  %3 = and i64 %0, 3
  br label %op_is_nonempty_list_fx_5

side_exit:                                ; preds = %EQ_next2,
                                          ;   %op_get_list_xxx_8,
                                          ;   %op_is_nonempty_list_fx_5
  %storemerge = phi i64* [ inttoptr (i64 140680233020480 to i64*),
      %%op_is_nonempty_list_fx_5 ], [ inttoptr (i64 140680233020416
      to i64*), %op_get_list_xxx_8 ], [ inttoptr (i64 140680233020416
      to i64*), %EQ_next2 ]
  store i64 %0, i64* %x0, align 8
```

```
  %4 = lshr i64 %fcalls, 2
  store i64 %4, i64* %struct.process.field.fcalls2, align 8
  %struct.process.field.htop = getelementptr %struct.process*
      %"process*", i64 %0, i32 0
  store i64* %htop, i64** %struct.process.field.htop, align 8
  %struct.process.field.stop = getelementptr %struct.process*
      %"process*", i64 0, i32 1
  store i64* %stop, i64** %struct.process.field.stop, align 8
  %struct.process.field.i = getelementptr %struct.process*
      %"process*", i64 0, i32 12
  %5 = ptrtoint i64* %i0 to i64
  %6 = ptrtoint i64* %storemerge to i64
  %7 = add i64 %5, -140680233020344
  %8 = add i64 %7, %6
  %9 = inttoptr i64 %8 to i64*
  store i64* %9, i64** %struct.process.field.i, align 8
  ret void

op_is_nonempty_list_fx_5:            ; preds = %op_move_xx_18,
                                     ;   %entry
  %GetList_src = phi i64 [ %15, %op_move_xx_18 ], [ %.pre, %entry ]
  %10 = and i64 %GetList_src, 3
  %11 = icmp eq i64 %10, 1
  br i1 %11, label %op_get_list_xxx_8, label %side_exit

op_get_list_xxx_8:                   ; preds =
                                     ;   %op_is_nonempty_list_fx_5
  %ListValue = add i64 %GetList_src, -1
  %12 = inttoptr i64 %ListValue to i64*
  %13 = load i64* %12, align 8
  store i64 %13, i64* %2, align 8
  %14 = getelementptr i64* %12, i64 1
  %15 = load i64* %14, align 8
  store i64 %15, i64* %1, align 8
  %16 = icmp eq i64 %13, %0
  br i1 %16, label %side_exit, label %EQ_next

EQ_next:                             ; preds = %op_get_list_xxx_8
  %17 = and i64 %3, %13
  %18 = icmp eq i64 %17, 3
  br i1 %18, label %op_move_xx_18, label %EQ_next2

EQ_next2:                            ; preds = %EQ_next
  %19 = tail call i32 @eq(i64 %13, i64 %0)
  %20 = icmp eq i32 %19, 0
  br i1 %20, label %op_move_xx_18, label %side_exit

op_move_xx_18:                       ; preds = %EQ_next2, %EQ_next
  store i64 %15, i64* %.phi.trans.insert, align 8
  store i64 %sub_fcalls, i64* %struct.process.field.fcalls2, align 8
  br label %op_is_nonempty_list_fx_5
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

EXAMENSARBETE A Tracing JIT Compiler for Erlang using LLVM
STUDENT Johan Fänge
HANDLEDARE Jörn Janneck (LTH), Haitao Li (Ericsson, Shanghai R&D)
EXAMINATOR Görel Hedin (LTH)

En tracing JIT-kompilator för Erlang

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Fänge**

Nästan alla moderna programspråk använder en interpretator – en flexibel och praktisk om än långsam lösning. Vi prövar ett enkelt sätt att kraftigt öka prestandan på Erlangs interpretator.

Det är vanligt att programspråk använder en interpretator för att köra ett program – exekvera programkod – istället för att kompilera direkt till maskinkod. Interpretatorn är ett litet program som läser programkoden i något bekvämt format, t.ex. som ren text, och sen utför det som står där direkt. Det finns flera fördelar med en interpretator:

- Lätt att skriva och underhålla.
- Lätt att göra dynamisk och flexibel.
- Behövs ingen komplicerad kompilering, kodgenerering eller länkning.
- Samma kod kan fungera på flera plattformar och processorer.

En interpretator är dock ofta långsam, och ett sätt att öka prestandan utan att förlora fördelar som plattformsoberoende är att kompilera till bytekod. Detta är simpla instruktioner, inte helt olikt en processors assemblerkod. Detta används av exempelvis Java, C#, Python och även Erlang. Det är lite som att baka klart en fryst pizza i ugnen – snabbt och smidigt och resultatet blir hyfsat, men att göra det från grunden i en riktig pizzaugn ger bättre kvalitet.

Programspråk som vanligtvis använder en interpretator utnyttjar ofta de möjligheter till flexibilitet det ger, vilket kan göra det nästintill omöjligt att kompilera ett sådant program direkt till maskinkod för att öka prestandan. Typexemplet på en besvärlig funktion är `eval`, som kan exekvera godtycklig programkod utifrån en textsträng.

En annan lösning för att nå liknande prestanda är då en Just-In-Time(JIT)-kompilator. Istället för att kompilera hela programmet till maskinkod i förväg, så väntar man med detta steg tills programmet exekverar och man t.ex. vet vilken plattform programmet kör på, eller vilken data det anropats med. Nu uppstår dock ett annat problem: det tar tid att kompilera och optimera kod. Typiskt börjar man därför i interpretatorn, och kompilerar först till maskinkod när en kodsnutt körs tillräckligt ofta för att det ska löna sig.

Ett lätt sätt att utöka en bytekodsinterpretator är genom en tracing JIT-kompilator. Grundprincipen är att man kör koden som vanligt i interpretatorn tills man stöter på en het loop, en kodsnutt som exekveras många gånger på rad, varpå man fortsätter att använda interpretatorn, men samtidigt också skriver ned vilka instruktioner man exekverar tills man kommer tillbaks till början av loopen. Detta bygger på insikten att den statistiskt vanligaste kodvägen typiskt också är den som med störst sannolikhet observeras.

Resultatet blir en trace, dvs en rak loopiteration som kan optimeras lätt och effektivt. Där programflödet kan avvika från den nedskrivna läggs speciella guard-instruktioner in som återgår till interpretatorn om dess villkor inte är uppfyllt.

I examensrapporten beskrivs våra erfarenheter av att ha utvecklat en tracing JIT-kompilator för programspråket Erlang, ett språk som används mycket på Ericsson där exjobbet påbörjades. I en utvärdering av prestandan på ett antal mindre benchmarks lyckades vi i vissa fall fyrdubbla exekveringshastigheten.

```
# original
names_by_id = [(23, "Kalle"),
  (1, "Eric"), (7, "Bo"), ...]

def find_name(wanted_id):
    for (id, name) in names_by_id:
        if id == wanted_id:
            return name
    return None
```

```
# trace
:trace_start
x = names_by_id.get_element(i)
if failed:
   goto :trace_abort
id = get_fst_tuple(x)
if id == wanted_id:
   goto :trace_abort
i = i + 1
goto :trace_start

:trace_abort
...
```

Exempel på en (påhittad) trace för Python