ErLLVM: An LLVM Backend for Erlang

Konstantinos Sagonas [1,2] Chris Stavrakakis [1] Yiannis Tsiouris [1]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
erllvm@softlab.ntua.gr

Abstract

This paper describes ErLLVM, a new backend for the HiPE compiler, the native code compiler of Erlang/OTP, that targets the LLVM compiler infrastructure. Besides presenting the overall architecture of ErLLVM and its integration in Erlang/OTP, we describe the changes to LLVM that ErLLVM required and discuss technical challenges and decisions we took. Finally, we provide a detailed performance evaluation of ErLLVM compared to BEAM, the existing backends of the HiPE compiler, and Erjang.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Retargetable compilers, Run-time environments; D.3.2 [*Language Classifications*]: Applicative (functional) languages, Concurrent, distributed, and parallel languages

General Terms Design, Languages, Performance

Keywords LLVM, HiPE, native code compiler, Erlang

1. Introduction

For about a decade now, the Erlang/OTP distribution contains the HiPE (High Performance Erlang) native code compiler [7] as an integrated component. By now the HiPE compiler is robust, handles the complete Erlang language and produces reasonably efficient code. It comes with backends for a variety of platforms, starting with SPARC V8+ that was the first architecture it supported, continuing with x86 [15] and AMD64 (x86_64) [12], PowerPC and PowerPC64, and various flavors of the ARM family of processors.

Each of these backends requires development and maintenance of a code base of significant size both in the compiler, which is written in Erlang, and in the code that interfaces with the Erlang Run-Time System (ERTS), which exists in C. Although some code can be shared between all these backends (e.g., the bulk of the code for the register allocators) or between backends which belong to the same family (e.g., x86 and x86_64), their maintenance and further development is not top priority for its original authors anymore. Perhaps it would be better if this work could be "outsourced" by using some compiler infrastructure which was more modern and more actively maintained.

One such infrastructure is that of LLVM. Its language-agnostic design makes LLVM currently quite popular both as a static or dynamic (Just-in-Time) compiler, and as a static program analyzer. LLVM seems quite popular nowadays and is used as a common infrastructure to implement a broad variety of statically and runtime compiled languages, e.g., the family of languages supported by GCC, Java and .NET [4], Python [18, 24], Ruby [13, 16], Haskell [22], as well as countless lesser known languages. It has replaced a broad variety of special-purpose compilers and has also been used to create a broad variety of new products (e.g., the OpenCL GPU programming language and runtime).

Wanting to experiment with LLVM in the context of Erlang, we embarked on a project, called ErLLVM, whose outcome is a new backend for the HiPE compiler that targets LLVM. We describe it in detail in this paper in order to document the current status of our implementation, measure its performance, describe a few technical challenges we faced and the tricks we used to overcome them.

Overview We review LLVM and its characteristics in Section 3. We describe in detail the design of ErLLVM, document the decisions we took, the technical challenges that we needed to address, and the changes that we made to LLVM's code base (Section 4). A detailed performance evaluation against BEAM, HiPE, and Erjang on a variety of programs is given in Section 5. But let us begin with a brief overview of HiPE and its existing backends.

2. HiPE

HiPE (High Performance Erlang) was an ASTEC (Advanced Software TEChnology competence center) project at Uppsala University during 1998–2005. The main goal of the HiPE project was to investigate ways of efficiently implementing concurrent programming languages using message-passing in general and the programming language Erlang in particular. One of the concrete outcomes of the HiPE project was the development of the HiPE native code compiler for Erlang. Since October 2001, the HiPE compiler is fully integrated in Erlang/OTP system and it is effectively *the* native code compiler for the Erlang language. For more information, see `http://www.it.uu.se/research/group/hipe`.

In this section, we will mostly present an overview of the architecture of the HiPE native code compiler and how it is integrated in the Erlang Run-Time System of Erlang/OTP focusing on issues that are relevant for the implementation of the new LLVM backend.

2.1 The HiPE Pipeline

Figure 1 outlines the architecture of the Erlang/OTP system when HiPE is enabled. (In the same figure, the box labeled LLVM shows where the new LLVM backend fits in HiPE's pipeline.) Although the compilation can start directly from source code, in most cases the code has already been compiled by BEAM, the bytecode compiler of Erlang/OTP. Thus, the compilation to native code starts by disassembling the bytecode and representing it in a symbolic form. Besides this symbolic BEAM representation, HiPE uses another two intermediate representations and ends up generating symbolic target-specific assembly. The representations used by HiPE are:

Erlang'12, September 14, 2012, Copenhagen, Denmark.

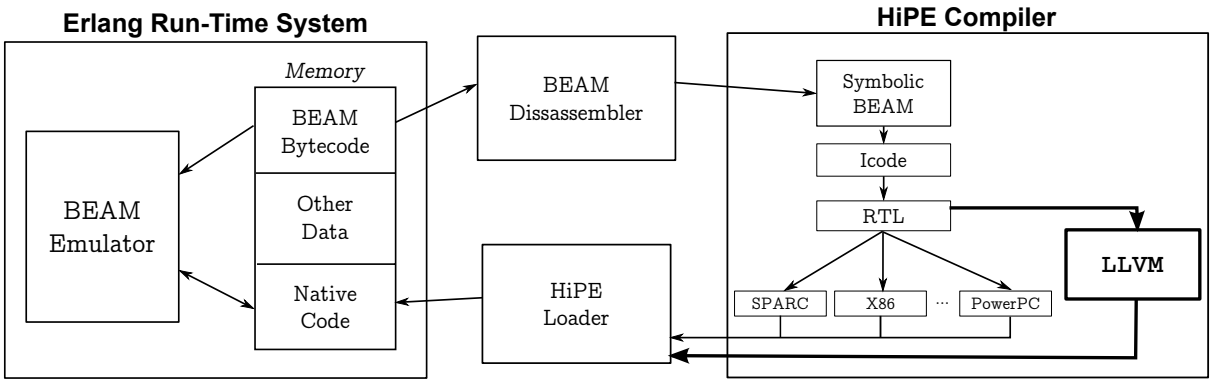

Figure 1: Architecture of the HiPE compiler extended with the LLVM backend.

Symbolic BEAM: This is a symbolic representation of the code that the bytecode compiler of Erlang/OTP has generated for execution in the interpreter of the BEAM virtual machine (VM). BEAM is a register-based VM and its bytecode has instructions that implement pattern-matching compilation, message-passing concurrency, process scheduling, automatic memory management of all memory areas (heap, stack, process mailbox, etc.), exception handling, type checking, bignum arithmetic, etc. All the optimizations that the bytecode compiler has performed are part of the symbolic BEAM representation that the native code compiler starts from. However, when this bytecode is loaded into the runtime system, the BEAM loader also performs aggressive instruction merging and specialization, which significantly expand the set of instructions and make the bytecode run considerably faster in the threaded interpreter of the VM. This latter kind of optimizations, i.e., those performed by the BEAM loader, are *not* part of HiPE's starting point.

Icode: Icode is an idealized Erlang assembly language with a minimal instruction set of only 16 instructions. The Icode intermediate representation (IR) assumes an infinite number of registers and an implicit stack. Registers are preserved around function calls and all bookkeeping operations, such as memory management and process scheduling, are implicit.

The symbolic BEAM is translated to Icode mostly one instruction at a time. However, some standard sequences of symbolic BEAM code are peephole-optimized into more efficient Icode sequences. Common operations, such as fetching an element from a tuple or switches that perform pattern-matching, are inline expanded into fetches and tests.

Temporaries are also renamed through conversion to *Static Single Assignment* (SSA) form [2] to avoid false dependencies between different live ranges. This form enables many optimizations at the Icode level, such as constant and copy propagation, constant folding, structure reuse, and dead-code elimination. Figure 2 displays the Icode of a simple tail-recursive function that computes the sum of all elements of a list. The `redtest` primitive operation implements process scheduling: it checks whether the process executing this code has exhausted its reduction steps and if so yields. Otherwise, there are tests whether `v3`, the first argument of the function after conversion to SSA form, is a cons (label 4) or nil (label 6), going to the appropriate basic block if the test succeeds. If `v3` contains some other term, a `function_clause` exception is raised (label 2).

RTL: RTL is a generic three-address *register transfer language*. RTL itself is target-independent, but the code is target-specific due to references to target-specific registers and primitive procedures. The RTL instruction set is RISC-like consisting of only 27 instructions. RTL has *tagged* registers for proper Erlang values (all Erlang terms are tagged in BEAM) and *untagged* registers for arbitrary machine values, such as addresses, raw integers or floating-point numbers. To simplify the garbage collector interface, function calls only preserve live tagged values.

```
1  lists:sum/2(v1, v2) ->
2  %% Info: ['Not a closure','Leaf function']
3  1:
4      v3 := phi({1, v1}, {3, v6})
5      v4 := phi({1, v2}, {3, v7})
6      _ := redtest() (primop)
7      goto 4
8  4:
9      if is_cons(v3) then 3 else 6
10 3:
11     v5 := unsafe_hd(v3) (primop)
12     v6 := unsafe_tl(v3) (primop)
13     v7 := '+'(v5, v4) (primop)
14     goto 1
15 6:
16     if is_nil(v3) then 5 else 2
17 5:
18     return(v4)
19 2:
20     v10 := function_clause
21     fail(error, [v10])
```

Figure 2: The Icode (in SSA form) of a `lists:sum/2` function.

In the translation from Icode to RTL a large number of operations (e.g., arithmetic, data construction, tests) are inlined. Data tagging and untagging are made explicit, data accesses and initializations are turned into loads and stores. Icode-level `switch` instructions for switching on basic values are translated into RTL code that implements these switches. Moreover, stack and exception handling code is expanded into explicit code. The RTL code is also represented in SSA form and optimizations like common subexpression elimination, constant propagation, constant folding, and dead code elimination are performed again in this form.

Figure 3 shows the RTL code that the HiPE compiler currently produces for the `lists:sum/2` function of Figure 2. Notice how the Icode primitive operations have been inlined, how untagging has been made explicit (e.g., lines 22, 23, and 25), how constant terms have been substituted by their internal representation (line 36), and how addition has been partially inlined

```
{lists,sum,2}(v19, v20) ->
;; Leaf function
;; Info: []
.DataSegment
.CodeSegment
L1:
    v21 <- v20
    v22 <- v19
    goto L2
L2:
    v23 <- phi({1, v21}, {11, v32})
    v24 <- phi({1, v22}, {11, v33})
    %fcalls <- %fcalls sub 1 if lt then L4 else L6
L4:
     <- suspend_0() then L6
L6:
    r25 <- v24 'and' 2 if eq then L7 else L8
L7:
    v26 <- [v24+-1]
    v27 <- [v24+7]
    r28 <- v26 'and' v23
    r29 <- r28 'and' 15
    if (r29 eq 15) then L13 else L12
L13:
    r34 <- v23 sub 15
    v35 <- v26 add r34
           if not_overflow then L11 else L12
L11:
    v31 <- phi({13, v35}, {12, v30})
    v32 <- v31
    v33 <- v27
    goto L2
L12:
    v30 <- '+'(v26, v23) then L11
L8:
    if (v24 eq -5) then L15 else L16
L15:
    return(v23)
L16:
    v36 <- atom_no('function_clause')
     <- erlang:error(v36)
    return(15)
```

Figure 3: RTL code (in SSA) produced by the Icode of Figure 2.

(fixnum addition happens in basic block L13 and, if bignum addition is required, control goes to basic block L12).

Symbolic target-specific assembly: This representation, used by the existing HiPE backends, is just a simple abstraction of the assembly of the target architecture (SPARC, x86, . . . , PowerPC) as Erlang terms. Symbolic assembly gets produced by translating RTL to it. It differs from RTL in that the code contains idioms which are target-specific (e.g., that the target architecture has a base register plus offset addressing mode).

2.2 The Existing HiPE Backends

The main phases of the existing backends are: 1) register allocation, 2) frame management, 3) code linearization, and 4) translation of the symbolic assembly to a loadable object. Let us describe them.

Register allocation. In this phase, the compiler attempts to map temporaries to actual machine registers. Every temporary that remains unallocated is mapped to a specific stack slot during the subsequent phase of frame management. The HiPE compiler has paid special attention to register allocation and provides a wide variety of allocators: an iterated register coalescing allocator [5], an optimistic register coalescing allocator [14], a Briggs-style graph coloring register allocator [1] and a linear scan register allocator [17, 19]. On register-poor architectures (e.g., x86) or in optimization level `o3` the allocator used by default is the iterated register coalescing. This allocator has been tuned over a number of years and is typically quite effective.

Frame management. After register allocation, stack frames are introduced to the code. This phase is responsible for:

- mapping spilled temporaries to stack slots, minimizing the size of the stack frame (i.e., by mapping temporaries whose life times do not overlap to the same stack slot), and rewriting uses of these temporaries as memory operands in the stack frame;
- adding code to the function prologue in order to check for stack overflow and setting up the call frame (the frame size and maximum stack usage are computed and taken into consideration);
- creating stack descriptors for each call site, describing the stack layout, the stack slots that correspond to live temporaries (using the result of the liveness analysis) and whether the call is in the context of a local exception handler; and
- generating code, at each tail call, to shuffle the actual parameters to the initial portion of the stack frame.

Code linearization. The previous phases operate on code that is in a *Control Flow Graph* (CFG) form. Before translating to native code, the CFG must be linearized by ordering the basic blocks and redirecting jump instructions accordingly. The linearization phase is responsible for performing this ordering while taking into account the likelihood of a conditional jump being taken or not, and the static branch prediction algorithm used in hardware. This is a crucial phase of the backend as it improves the efficiency of the generated native code.

Assembling. Finally, a custom assembler converts the symbolic assembly to binary code and produces a loadable object (i.e., an appropriate Erlang term) with the machine code, the constant data, the symbol table, and the external references needed to be patched for the runtime system. Notice that the assembling phase is also an *in-memory* phase and no intermediate files are created at any step in the compilation pipeline.

2.3 Pros and Cons of the Existing HiPE Backends

With the exception of offering a wide selection of register allocators and of performing stack minimization, which pays off when garbage collection is triggered, the backends of HiPE are relatively simple. Instruction selection is standard and none of the backends performs instruction scheduling. Also, the HiPE backends by default generate native code for the lowest common denominator of the target family. For example, the x86 backend still generates x87 code for floating point operations by default even though nowadays there are probably very few x86-based machines in operation that do not support SSE2 (Streaming SIMD Extensions 2). As another example, the ARM backend of HiPE has never been updated to generate floating point instructions; till recently, most ARM processors did not come with hardware support for floating point.

In short, we expect that by outsourcing to LLVM: instruction selection, instruction scheduling, and the task of detecting the presence of special hardware and optimizing for it, the generated native code can be better than the one currently produced by HiPE backends.

3. LLVM

LLVM is an open source compiler infrastructure written in C++ that started at the University of Illinois at Urbana-Champaign by Chris Lattner [9]. Nowadays, LLVM is a collection of modular and reusable state-of-the-art compiler and toolchain technologies, that

can be used to create static compiler backends, Just-in-Time (JiT) compilers or mid-level and optimization analysis tools on a compiler pipeline. It provides a source- and target-independent optimizer that operates *only* on the LLVM Intermediate Representation and targets one of the available code generators to produce native machine code for a variety of architectures.

LLVM's novelty comes from the following features:

- The organization of the optimizer as a set of well-defined *libraries*.

  The compilation pipeline comprises of more than 100 distinct analysis and optimization passes. These optimizations include many of the classic compiler optimizations found in compiler text books. Each pass is run on the input and, after verifying a specific transformation, updates the code, preparing it for the next optimization pass or the code generator. The compiler designer is able to select which passes should be executed along with their order, or even implement new ones that leverage these libraries and construct the intended pipeline.

- The support for lifelong program analysis through *Link-Time Optimization* (LTO) and *Install-Time Optimization* (ITO).

  Link-Time Optimization is used to perform optimization (like inlining) across file boundaries. With LTO enabled, the compiler emits LLVM bitcode (cf. Section 3.1) instead of native code to the object file, and delays code generation to link time. The LLVM Linker is responsible for identifying that the object files contain LLVM bitcode, for loading them in memory and for triggering the optimizer over the aggregate and, then, the code generator.

  Install-Time Optimization is based on the same idea and delays code generation even later. By delaying, for example, instruction choice, scheduling, code layout, and other aspects of code generation, more appropriate choices can be made to better leverage the underlying hardware.

- The design of the LLVM Intermediate Representation as a complete code representation that enables the aforementioned features.

ErLLVM does not take advantage of lifelong program analysis, since Erlang supports language-level *Dynamic Software Updating* (DSU) (also known as "hot-code loading") which currently restricts the inter-procedural optimizations; hence, we will not expand on this feature now. Instead, we will examine more thoroughly the LLVM IR, as this serves as our interface with the LLVM.

3.1 The LLVM Intermediate Representation

The *LLVM Intermediate Representation* (IR), or *LLVM assembly*, is a high-level portable assembly language, providing abstraction between native assembly and source language. It is basically a common, low-level, code representation in SSA form. Some of its innovative features include: a strict, language-independent type system that exposes the primitives commonly used to implement high-level language characteristics; an instruction for typed address arithmetic (`getelementptr`); simple mechanisms that can be used to implement garbage collection and exception handling in a uniform manner. LLVM IR was designed to be a "universal intermediate representation" able to express many different characteristics and host various optimizations for arbitrary high-level programming languages.

Types need explicit handling from the programmer and no type inference is performed on an instruction. The type system was thoroughly designed to enable a broad class of *high-level* transformations on *low-level* code without the need for extra analysis on the side. Furthermore, type-mismatches can be used to detect errors in optimizations by the LLVM consistency checker.

LLVM provides three different, yet equivalent, code representations: an in-memory compiler IR, an on-disk binary format, known as *LLVM bitcode* (suitable for compact saving and fast loading by a Just-in-Time compiler), and a human-readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while also providing a natural means to debug and visualize the transformations.

LLVM programs are composed of *modules*, each of which is a translation unit of the input program. Figure 4 contains the optimized LLVM module of `lists:sum/2` example of Figure 3. Each module consists of metadata, global variables, external symbol definitions and function definitions. Metadata include definitions of the endianess, the pointer size and the alignment of the data layout. Global variables define regions of memory allocated at compile-time rather than at runtime. They are pointers to data with global scope and are prefixed with the `@` symbol, similar to functions (e.g., `@lists.sum.2`). This should be compared to the local temporaries (virtual registers and labels) spotted within a function scope and denoted with the `%` prefix, e.g., `%1`–`%68`. The first `N` virtual registers, i.e., `%1`, `%2`, `%3` and `%4`, are used for the incoming arguments. External declarations, such as `@atom_function_clause`, define symbols that may be used in the current module but are meant to be defined during linkage. Comments begin with a `;` and go until the end of the line.

A function definition contains a list of *basic blocks*, forming the Control Flow Graph (CFG); in our example code there are many such blocks. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and explicitly ends with a terminator instruction, such as a branch (e.g., a `br`) or a function return (`ret`). The first basic block in a function is not allowed to have predecessor basic blocks (i.e., there cannot be any branches to the entry of the function). There is a straightforward correspondence of the LLVM blocks in Figure 4 with the RTL blocks in Figure 3, e.g., `%21` corresponds to block `L4` (suspend block), `%31` to `L7` (data untagging), `%42` to `L13` (fixnum addition), etc.

All operations are explicitly annotated with types, e.g., in line 14 a 64-bit integer value (`-5`) is stored in a consistent pointer variable (`i64* %5`) already allocated on the stack (notice the `%5 = alloca i64` instruction on line 11). Another example is that of explicit pointer arithmetic performed using `add`, then `inttoptr` (for explicit type conversion) and, then, `load` which can be seen in lines 27–29. The number of arguments and return values in calls differ from the corresponding ones in Figure 3; this is closely related with the `cc11` annotations in function definitions and calls, and will be discussed later on in Section 4.2.

3.2 Garbage Collection in LLVM

LLVM provides garbage collection built-ins (*intrinsically*) in the intermediate representation and a framework for compile-time code generation plugins. These facilities should generically support various popular garbage collection algorithms, such as semi-space, mark-and-sweep, generational, reference counting, incremental, concurrent and cooperatives, without providing an actual implementation for them.

Using the plugins one can generate code and data structures that conform to the *binary interface* specified by the language's *runtime library* (hosting the garbage collector): denote safe-points, compute and print stack maps, and use read/write barriers.

The available intrinsics are:

`llvm.gcroot:` Declares the existence of a GC root to the code generator. The potential root *must* be a stack allocated value, i.e., a virtual register previously declared with an `alloca` in-

```
1  declare void @llvm.gcroot(i8**, i8*)
2  declare cc11 { i64, i64, i64 } @suspend_0(i64, i64)
3  declare cc11 { i64, i64, i64 } @erlang.error.1(i64, i64, i64)
4  declare cc11 { i64, i64, i64 } @bif_add(i64, i64, i64, i64)
5  declare cc11 { i64, i1 } @llvm.sadd.with.overflow.i64(i64, i64)
6
7  @atom_function_clause = external constant i64
8
9  define cc11 {i64, i64, i64} @lists.sum.2(i64, i64, i64, i64)
10     nounwind noredzone gc "erlang_gc" {
11   %5 = alloca i64
12   %6 = bitcast i64* %5 to i8**
13   call void @llvm.gcroot(i8** %6, i8* @gc_metadata)
14   store i64 -5, i64* %5
15   ; More gcroot declarations
16   ...
17   br label %11
18 ...
19 ; <label>:21                          ; preds = %11
20   %22 = call cc11 { i64, i64, i64 }
21         @suspend_0(i64 %13, i64 %12) nounwind
22   %23 = extractvalue { i64, i64, i64 } %22, 1
23   %24 = extractvalue { i64, i64, i64 } %22, 0
24   br label %25
25 ...
26 ; <label>:31                          ; preds = %25
27   %32 = add i64 %28, -1
28   %33 = inttoptr i64 %32 to i64*
29   %34 = load i64* %33
30   store i64 -5, i64* %7
31   %35 = add i64 %28, 7
32   %36 = inttoptr i64 %35 to i64*
33   %37 = load i64* %36
34   store i64 %37, i64* %5
35   %38 = load i64* %9
36   %39 = and i64 %34, 15
37   %40 = and i64 %39, %38
38   %41 = icmp eq i64 %40, 15
39   br i1 %41, label %42, label %52
40
41 ; <label>:42                          ; preds = %31
42   %43 = add i64 %38, -15
43   %44 = call { i64, i1 }
44         @llvm.sadd.with.overflow.i64(i64 %34, i64 %43)
45   %45 = extractvalue { i64, i1 } %44, 0
46   %46 = extractvalue { i64, i1 } %44, 1
47   br i1 %46, label %52, label %47
48 ...
49 ; <label>:52                          ; preds = %42, %31
50   store i64 -5, i64* %9
51   %53 = call cc11 { i64, i64, i64 }
52         @bif_add(i64 %27, i64 %26, i64 %34, i64 %38) nounwind
53   %54 = extractvalue { i64, i64, i64 } %53, 1
54   %55 = extractvalue { i64, i64, i64 } %53, 0
55   %56 = extractvalue { i64, i64, i64 } %53, 2
56   %57 = load i64* %5
57   br label %47
58 ...
59 ; <label>:60                          ; preds = %58
60   %61 = load i64* %9
61   %62 = insertvalue { i64, i64, i64 } undef, i64 %27, 0
62   %63 = insertvalue { i64, i64, i64 } %62, i64 %26, 1
63   %64 = insertvalue { i64, i64, i64 } %63, i64 %61, 2
64   ret { i64, i64, i64 } %64
65
66 ; <label>:65                          ; preds = %58
67   %66 = ptrtoint i64* @atom_function_clause to i64
68   %67 = call cc11 { i64, i64, i64 }
69         @erlang.error.1(i64 %27, i64 %26, i64 %66) nounwind
70   %68 = insertvalue { i64, i64, i64 } %67, i64 15, 2
71   ret { i64, i64, i64 } %68
72 }
```

Figure 4: Partial LLVM assembly output of the Optimizer for `lists:sum/2` of Figure 3.

struction. A usage example can be seen in the entry block of the code (lines 11–14) in Figure 4.

`llvm.gcread`/`llvm.gcwrite`: Identifies reads (writes) of references from (to) heap supporting garbage collector implementations that require read or write barriers.

These intrinsics can be lowered in actual code in the way that is defined in the GC plugin.

3.3 Exception Handling in LLVM

The exception handling interface of LLVM has recently been redesigned in order to conveniently support the two most common schemes of exception handling utilised by programming languages: the *dynamic registration* and the *table-driven* approach.

The first scheme is supported through what is called *Setjmp/-Longjmp* (SJLJ) exception handling [10]. LLVM implements this by providing appropriate intrinsics (`llvm.eh.sjlj.*`) for building, accessing and removing the unwind frame contexts at runtime. This approach is more compact in terms of space and faster for handling a thrown exception, but adds execution overhead on normal control flows. Thus, the second scheme is generally preferred.

The *Itanium ABI Zero-cost* exception handling [10] is the most well-known implementation of the table-driven scheme, also used by many production-quality C++ compilers [3]. It defines a methodology for creating static tables, called *Exception Handling Tables*, at compile- and link-time which relate program regions with exception handlers. In case of a thrown exception, the runtime can use these tables to determine how to unwind the stack and handle the exception. Thus, this specification adds "zero-cost" to the normal execution of an application.

The IR provides three instructions related to exception handling:

`invoke`: Causes control to transfer to a specific function, like a normal `call`, with the possibility of the control flow to return to either a "normal" label or an "exception" label.

`landingpad`: Specifies a basic block where an exception lands; saves the exception structure reference; then, proceeds to select the catch block that corresponds to the type info of the exception object. It can be used to perform a "catch", a "cleanup" or a "filter" on the exception object.

`resume`: Resumes propagation of an existing (in-flight) exception whose unwinding was interrupted with a `landingpad` instruction.

4. ErLLVM

Important requirements in the design of ErLLVM have been to fit as easily as possible within the existing pipeline of the HiPE compiler and maintain Application Binary Interface (ABI) compatibility in order to enable interoperability with the rest of HiPE's infrastructure. As depicted in Figure 1, ErLLVM is placed after the RTL passes, just like the other backends of HiPE. This decision is based on the fact that RTL aims to represent Erlang code in a form that is as low-level as possible while still being hardware-independent; in RTL all high-level characteristics of Erlang, such as exception handling and garbage collection, have been explicitly expanded to low-level code.

Although the RTL and LLVM intermediate representations have many similarities, and the mapping from one to the other is almost straightforward, the fact that Erlang code is executed in a Managed Runtime Environment (MRE) introduces several obstacles in producing efficient code using the LLVM infrastructure. Most of these are related to the fact that the Erlang Run-Time System (ERTS) provides automatic memory management by employing a precise garbage collector. In this section, we will discuss these issues but let us first go through some details of the *new* HiPE pipeline.

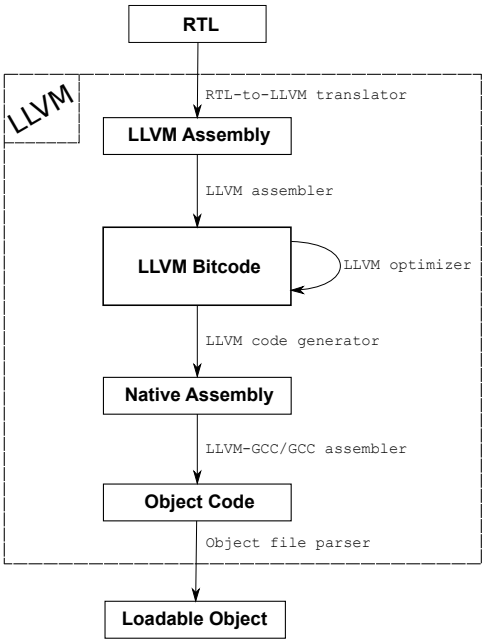

Figure 5: The LLVM component of HiPE's pipeline.

4.1 The ErLLVM Pipeline

As we have already discussed in Section 2.1, the original pipeline of the HiPE compiler produces symbolic native assembly for the target architecture and, after optimizing the code, uses HiPE's custom assembler to generate binary code that will be loaded to the Erlang Run-Time System by the HiPE loader. HiPE's assembler operates specifically on the in-memory symbolic assembly representation generated by HiPE. Additionally, along with assembling, it gathers all the information that is required by the loader to be able to load the binary code. This information includes:

- the offsets of the compiled functions and closures;
- the function calls with their stack descriptors and addresses of exception handlers;
- the Erlang terms that need to be interned in tables, such as constants and atoms; and
- the jump tables for switch statements.

The new LLVM backend takes RTL as input and aims at producing native code and all the anticipated loading information using the LLVM infrastructure. In this way we do not need to modify the HiPE loader. The ErLLVM pipeline, illustrated in Figure 5, consists of the following phases:

Translation of RTL to LLVM assembly. Direct mapping of the RTL code representation to the corresponding LLVM assembly. Some information (e.g., the values of the jump tables) is collected to be used at load time. The LLVM assembler is triggered in order to translate the human-readable LLVM assembly to binary format, the *LLVM bitcode*, which is more efficient for the optimizer.

Compilation and optimization of the LLVM assembly. LLVM's optimizer and code generator are employed to create a file containing the optimized native assembly.

Assembling. An object file is created from the native assembly using an available assembler, e.g., the `llvm-gcc` or the `gcc` assembler.

Object file parsing. As already mentioned, the HiPE loader does not take as input an object file. Instead, its input is an Erlang record containing a binary with the native code and all information necessary for loading this code. ErLLVM parses the object file and uses the information collected in the first phase of this list, in order to construct the aforementioned record.

Compiling RTL to LLVM Assembly

As already mentioned, both the RTL and the LLVM IR consist of a low-level instruction set and memory model, and therefore all aspects of RTL can be directly mapped to LLVM assembly. The primary difference is that LLVM assembly includes a static type system, whereas RTL is generally untyped. Fortunately, mapping an untyped language to a typed one is not difficult, since when the type of a variable cannot be inferred from the instruction in which it is used, we can safely consider it to have the type `word` (i.e., the size of a generic pointer) of the target architecture.

This compilation phase is done per function, as functions are the compilation units of the HiPE compiler. Each instruction is mapped to one or more symbolic LLVM instructions which are internally represented as Erlang records. Then, the human-readable LLVM assembly for each function is dumped in a file and serves as the point of interaction with the LLVM framework. We chose a human-readable representation for this interaction because: (a) this representation is more suitable for debugging, and (b) in the beginning of ErLLVM's development, there were no Erlang bindings available for the LLVM API.

4.2 Calling Convention

Different languages use different calling conventions to describe how the caller and the callee interact during a call. A calling convention specifies: (a) where the parameters and the return values are placed, (b) in which order, (c) which are the caller-/callee-saved registers, and (d) how the task of setting-up and restoring the stack after a function call is divided between the caller and the callee (e.g., the callee pops the arguments off the stack).

LLVM can support multiple calling conventions, by augmenting each function call with a calling convention annotation. HiPE, though, uses a custom calling convention which does not match with any of those provided by LLVM. So, we extended LLVM by implementing a new calling convention (`cc11`) which describes the scheme used by the HiPE compiler for the X86 and the X86_64 architectures[1]. This was the first patch we had to apply to LLVM.

Precoloured Registers

In RTL most registers are abstract (*virtual*) and independent of the target architecture. However, a small subset of them, that includes frequently accessed components of the ERTS, is mapped to actual hardware registers for performance reasons. HiPE implements this by treating these architecture registers as unallocatable; such registers are: the Native Stack Pointer (`NSP`), the Process Control Block Pointer (`P`), and the Heap Pointer (`HP`). The `NSP` register contains a pointer to the top of the Erlang runtime stack. Similarly, `HP` contains a pointer to the next free memory location in the heap. `P` points to a location in memory storing useful information about the Erlang process that is currently running. These registers are called *precoloured* (or *pinned*) in the following.

The LLVM assembly is designed to be strictly target-independent and offers no way of interacting with the architecture. Pinning precoloured registers to physical registers would require explicit control over register allocation, which is something that is not currently offered by LLVM. Fortunately, the same problem was attacked by

[1] Detailed information about the calling conventions used in HiPE appears in `$OTP_ROOT/erts/emulator/hipe/hipe_$ARCH_abi.txt` text files.

```
define f (%arg1) {
  ...
  res = call g (%arg1, %tmp);
  ...
  return 0;
}
```

↓

```
define cc11 f (%HP, %P, %arg1) {
  ...
  {%HP'', %P'', res} =
      call cc11 g (%HP', %P', %arg1, %tmp);
  ...
  return  {%HP''', %P''', 0};
}
```

Figure 6: A simplified example of the transformation performed by the backend in order to achieve register pinning using the new calling convention in LLVM.

David Terei and Manuel Chakravarty when they implemented an LLVM backend for the Glasgow Haskell Compiler (GHC) [22] based on suggestions by Chris Lattner[2].

Figure 6 shows how we achieved to place our "special" virtual registers to specific physical ones by leveraging the new calling convention (described in Section 4.2). We properly defined the calling convention to: pin the first n arguments to registers (where n is the number of precoloured registers) and, also, pin the first n return values to the same physical registers. Then, our backend translates each function call to a new one, which has n *extra* parameters and return values, and uses this calling convention to guarantee that the precoloured registers reside in the corresponding physical ones. The HiPE calling convention defines that `NSP` resides in `SP` (e.g., `ESP` for X86 and `RSP` for X86_64); LLVM, also, uses `SP` for the same reason. Hence, the `NSP` virtual register is not included in the transformation. Concrete examples of extracting return values *from* a call and returning values *to* the caller can be seen in lines 20–23 (block `%21`) and lines 60–64 (block `%60`), respectively, of Figure 4.

With this transformation, the precoloured registers will have the correct values on function entry and return. Throughout the body of the function these registers are handled like any other virtual register allowing, for example, the register allocator to spill them to the stack when there is high register pressure. However, this is not a problem as precoloured registers should, in fact, be pinned to hardware registers only upon entry and exit of a function, as these are the points of interaction with the runtime system. Actually, this approach should offer better performance since there are more registers available for the register allocator, and, thus, should produce more efficient code. However, this is not the case for the reasons that we will go through in Section 5.

4.3 Garbage Collection

Erlang, as most functional programming languages, relies on precise garbage collection (GC) for automatic memory management. A precise garbage collector attempts to reclaim memory occupied by objects no longer in use by the program by tracing pointers to heap objects. Precise GC requires support from the runtime system and may have great impact on performance. As many other compilers, HiPE uses *stack maps* (or *stack descriptors*) computed at each safe point in order to expose this information to the garbage collector.

[2] The discussion can be found in http://nondot.org/sabre/LLVMNotes/GlobalRegisterVariables.txt.

```
1  Entry:
2     ;; In the entry block of the function,
3     ;; allocate stack space for virtual register %X.
4     %X = alloca i64*
5
6     ;; Tell LLVM that the stack space is a stack root.
7     %tmp = bitcast i64** %X to i8**
8     call void @llvm.gcroot(i8** %X, i8* null)
9     ;; Store the 'nil' value into it, to indicate that
10    ;; the value is not live yet. "-5" is the tagged
11    ;; representation of 'nil'.
12    store %i64 -5, %64** %X
13    ...
14    ;; "CodeBlock" is the block corresponding to the
15    ;; start of the scope of the virtual register %X.
16 CodeBlock:
17    store i64 %some_value, i64** %X
18    ...
19    ;; As the pointer goes out of scope, store
20    ;; the 'nil' value into it, to indicate that
21    ;; the value is no longer live.
22    store %i64 -5, %64** %X
```

Figure 7: LLVM assembly for handling a GC root.

Currently, LLVM supports precise garbage collection by the `llvm.gcroot` intrinsic we described in Section 3.2. A compiler author can write an appropriate GC plugin to export a stack map in the form that is required by the runtime system. Hence, we added a GC plugin in the LLVM source code — this was our second set of patches — in order to be ABI compatible with ERTS.

The problems arise from the fact that LLVM strictly demands the variables, which will be used for hosting GC roots, to be allocated in the stack in the entry block of a function and, thus, are considered live for the entire function[3]. With this approach, the root property is attributed to *stack variables* rather than values/virtual registers. In Figure 7 we present the LLVM assembly that is generated in order to handle a root in the `%X` virtual register. In the current scheme, it is the responsibility of the frontend to copy values that contain garbage collectable objects to stack variables. Also, the frontend has to explicitly track down the liveness information of these variables and save a value that is not traceable by the garbage collector (the `nil` value) in order to denote them as *dead*.

This approach obviously leads to suboptimal code since the LLVM code generator, that has more information about all virtual registers, should be responsible for spilling and reloading roots on and from the stack around safe points with the goal of creating minimal stack maps by re-using stack locations for variables with non-overlapping lifetimes. Even worse, saving `nil` to denote that a stack variable does not contain a live pointer has a runtime overhead (i.e., the garbage collector traverses *all* stack slots and considers those that store `nil` as dead) and corrupts the static property of stack maps.

The latest version of ErLLVM tries to minimize this overhead by doing a liveness analysis on the frontend for RTL registers that are potential roots and conservatively mapping those with non-overlapping lifetimes to the same stack variable. But still, while improving the performance slightly, this solution cannot produce code as efficient as the existing backends of HiPE. Furthermore, the

[3] While the stack variables/stack slots are considered live during the lifetime of the function, the values that "inhabit" them (i.e., the actual pointers on the heap) may not be, for example if the compiler explicitly stores a value untraceable by the garbage collector in the corresponding stack slot.

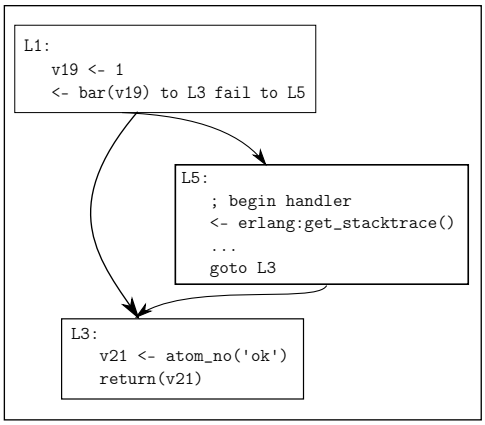

Figure 8: RTL control flow graph of a function calling `bar/1`, protected with an exception handler (the basic block with label `L5`).

translation of RTL to LLVM becomes more complicated without the anticipated reward.

4.4 Exception Handling

Exception handling in HiPE is implemented by adding an extra label to call instructions, if they are in the scope of some exception handler. A local exception handler is represented by the basic block that implements it. So, exceptions in HiPE's control flow graph are just edges between a basic block that ends with a call and a basic block that implements the handler. An example of a CFG that includes a call that is protected with an exception handler can be seen in Figure 8.

The address of the call, along with the address of the exception handler, are information that is exported to the runtime system through the stack descriptors. In this way there is no runtime cost for setting-up an exception handler since, when an exception is thrown, the stack map is used for unwinding the stack and transferring the control to the corresponding handler.

We used the Itanium ABI exception handling of LLVM (cf. Section 3.3) in order to implement the corresponding functionality. We found very useful for this the `invoke` instruction; this instruction operates as a standard call, with the only difference being that it creates a correlation between the call and the two labels, which is used at runtime. This correlation is exposed in the object file in the form of Exception Handling Tables in the `.gcc_except_table` section of an ELF object file.

Since these Exception Handling Tables contain all the necessary information which is required by the runtime system, ErLLVM just reads them from the object file and creates the corresponding stack descriptors, in the format expected by the HiPE Loader (see Section 4.1).

4.5 Explicit Stack Management

The HiPE compiler manages the Erlang call stack separately from the standard C stack[4]. During the translation from the high-level Erlang code to the low-level RTL code, the stack manipulations are made explicit. Allocation in this stack is done in small pieces instead of big chunks in order to optimize stack usage, especially for functions that do not make any call (*leaf functions*). As described in Section 2.2, the compiler prepends code to each function, which checks for stack needs and, if the function does not have enough space to execute, calls a BIF to allocate more space. This prologue must follow the frame initialization phase because the size of the function's frame must be known in order to perform the stack check. Thus, it can only be inserted by the compiler that setups the frame.

In LLVM there is a special post register allocation (post-ra) pass named *Prologue/Epilogue Insertion* (PEI) which is responsible for finalizing the function's frame layout, saving callee-saved registers, and emitting proper prologue and epilogue code for the function. So, we needed to modify the LLVM source code again — this was our third patch — in order to extend this pass and force the LLVM code generator to insert the special prologue code when compiling HiPE functions.

[4] An exception is made for primitive built-in functions (BIFs).

5. Evaluation

Next, we will report on the performance of the new LLVM backend and compare it (on x86_64 and x86) with BEAM, the existing HiPE backends, and, whenever possible, with Erjang.

The Erjang system [23] is an implementation of Erlang based on Java Virtual Machine (JVM) which compiles BEAM bytecode into JVM bytecode and then relies on *Just-in-Time* compilation to native code for speed. Roughly, what happens is that Erjang loads Erlang's binary `.beam` file, converts it into Java's `.class` file format, and loads it into the JVM. It also implements a BEAM interpreter that uses a *shared heap* [6] memory model for processes. (This means that in Erjang messages are not copied between Erlang processes.) The main benefit of Erjang is that it takes advantage from a virtual machine with very mature implementation technology, thanks to the vast amount of engineering effort that has been put into it, and a JiT compiler that is able to perform aggressive optimization and inlining of the code of "hot spots".

5.1 Benchmarks

We wanted a compiler benchmark suite that consisted of programs that run for a reasonable amount of time (more than a couple of seconds) so that the impact of low-level optimizations could be reliably measured, but less than 10 minutes in order to have easily reproducible results. Thus, for this paper, we created a simple benchmarking infrastructure [21] with benchmarks from two groups:

Micro-benchmarks These are mostly common benchmarks used in the past to evaluate the existing HiPE backends. Some of them (barnes) are floating-point intensive, some of them (life and smith) are concurrent, and the others are sequential programs that manipulate integers (tak), strings (prettypr, huff, and yaws_html), binaries (decode), and lists (length, length_u, nrev, and qsort).

We also included in this group a sequential program, orbit_seq, that solves the *orbit problem*: given a space X, a list of generators f_1,...,$f_n : X \to X$ and an initial vertex $x_0 : X$, the goal is to compute the least subset Orb of X such that Orb contains x_0 and is closed under all generators. (We note that orbit is far from a micro-benchmark; it consists of several Erlang modules of a total of about $1,000$ lines of Erlang code. It also comes with parallel and distributed versions of solving the same problem.)

Shootout This group consists of concurrent programs (binary-trees, chameneos-redux, fannkuch-redux, fasta, mandelbrot, n-body, pidigits, spectral-norm and thread-ring) that come from the so called "The Computer Language Benchmarks Game" [20]. They were created to compare performance across a wide variety of programming languages and their different implementations. They are small, yet non-trivial, concurrent programs that run for a considerable amount of time.

For BEAM, HiPE and ErLLVM, we have executed each benchmark five times in the same VM and kept the *median* value in order to remove the noise from the measurements. Measuring Erjang

performance was a bit more complicated: since this system needs to warm up, we run each benchmark four times in order to give the JVM a chance to find the hot spots and optimize them and with a small pause of one second between runs to let its JiT compiler be scheduled. We then kept the *average* value of the *next* two runs. We decided to report both the time for the first run ("Erjang - 1st") and the average time for the fifth and sixth runs ("Erjang - 5th"). The difference between the two times shows the effect, presumably positive, that the Java JiT compiler has.

The experiments for the x86_64 (AMD64) architecture were conducted on a machine with four Intel Xeon E7340 CPUs (2.40 GHz) quad-cores, having a total of 16 cores and 16 GB of RAM. For the x86 measurements we used a dual-core Intel Pentium D CPU (3.00 GHz) with 1 GB of RAM. Both machines run Debian GNU/Linux 2.6.32-5 and use GCC 4.4.5.

5.2 Runtime Performance

Figure 9 shows the results of a runtime comparison for the x86_64 (AMD64) architecture in the form of speedups from BEAM and Figure 10 contains performance results for x86. As can be seen, on average, code produced by the ErLLVM is two to three times faster than BEAM bytecode and almost as fast as the native code produced by HiPE. Note that on three of the benchmarks (mandelbrot, smith and tak), the speedup is considerably higher than this average (esp. on x86). On the other hand, on concurrent benchmarks such as thread-ring, pidigits and life, which spend the majority of their execution in Erlang BIFs implemented by the Erlang runtime system in C, the speedups are non-existent.

Overall, it is clear that the new LLVM backend has not managed to outperform the code produced by the existing HiPE backends, as we initially hoped it would do, given the current expectations from LLVM. Still, we think that these results are quite promising taking into account the years of development that HiPE has had. On a more positive note, as can be observed from the performance difference between length and its "loop unfolded" version length_u, especially on x86_64, ErLLVM prefers code with big code blocks where LLVM's optimizer can be more effective.

Native code, either produced by HiPE or ErLLVM, seems to perform as least as well as Erjang in the majority of benchmarks. Erjang is particularly fast in benchmarks containing a lot of floating point arithmetic and huge message passing, like chameneosredux and nbody. Notice, however, that the additional speed up that JiT compilation gives to Erjang is in most cases pretty small.

The evaluation of a compiler backend often focuses exclusively on runtime performance. However, we believe that in the case of a compiler for an industrially relevant language there are more factors that must be taken into consideration: compilation time, code size, and the backend code maintainability. So, we will try to evaluate our backend in a more complete way.

5.3 A Deeper Look into ErLLVM's Runtime Performance

To further inspect the performance of ErLLVM we examined the impact of different optimization levels (`-O1`, `-O2` and `-O3`) provided by the LLVM target-independent optimizer (`opt`). In Table 1 we explore this impact for both the X86 and the X86_64 backends. Surprisingly, some benchmarks experienced a slowdown when an optimization level was used, while most of them showed a small speedup. The first row of each sub-table presents the average (Avg.) of *slowdowns*, while the second row depicts the corresponding mean value of benchmarks exhibiting *speedups*; the third line presents the average value of all benchmarks. It is obvious that ErLLVM's performance is only improved slightly when compiled with the `-O2` optimization group ($\approx 5\%$), while `-O3` produces minor further improvement. The dearth of improvement is somewhat disappointing given the large number of optimizations performed by LLVM compared to the other HiPE backends. This also shows that the code generated for Erlang via Icode and RTL is not the "typical" code that compilers for C-like languages generate.

	O1	O2	O3
Avg. of Slowdowns	0.96	0.97	0.98
Avg. of Speedups	1.09	1.11	1.09
Total Avg.	1.02	1.03	1.04

(a) x86

	O1	O2	O3
Avg. of Slowdowns	0.96	0.93	0.93
Avg. of Speedups	1.07	1.10	1.13
Total Avg.	1.05	1.06	1.08

(b) x86_64

Table 1: The impact of different optimization levels in the LLVM optimizer against no optimizations used (`-O0`/`-OX`).

The biggest drawback of ErLLVM, that has a great negative impact on performance, is the code that supports precise garbage collection, as explained in Section 4.3. The way that is imposed by the LLVM for handling garbage collection roots significantly reduces runtime performance, as the static property of root liveness shifts to memory stores during runtime. Even more the garbage collector must traverse a bigger stack, as currently LLVM does not perform any form of stack minimization for garbage collection roots in order to allocate two roots to the same stack offset if their live ranges do not overlap. To get some indication of how much the code could be improved by stack minimization, we performed some manual optimization of root handling in the produced code and we got up to 20% improvement.

We also believe that the minor improvement achieved with different optimization levels is due to the nature of the compiled code. This can also be seen by the small speed up that JiT compilation gives to Erjang. It is common for Erlang modules to consist of small and simple functions; the native code usually follows a pattern of a few primary operations and low register usage but many function calls and high memory traffic. Additionally, because of the dynamic typing of Erlang, even primary operations may involve BIF calls, for example Erlang uses arbitrary-sized integers (bignums) and integer arithmetic is performed with a built-in function (when needed). Finally, hot-code loading restricts the effect of inter-procedural optimizations; the unit of compilation in HiPE is a single function and, thus, each function is compiled independently in ErLLVM. Therefore, no optimization can occur even between functions in the same module. Due to all these reasons, most of LLVM's optimization passes have no significant effect currently. We believe that there is work to be done in the future in order to take advantage of all of the optimizations that LLVM offers.

5.4 Binary Code Size

We used two applications of the Erlang/OTP system: the Standard Library (`stdlib`) and the HiPE compiler (`hipe`), comprised of 79 and 196 modules respectively, in order to compare the binary code sizes that are generated by ErLLVM to those generated by HiPE both for x86_64 and x86. Table 2 displays the aggregate results for these applications.

The LLVM backend produces 15–20% larger binaries than the existing HiPE backends. This is imposed mainly by the the way it handles garbage collection roots and the need for explicitly marking roots liveness with store instructions. Secondly, LLVM tries to optimize the binary code to better fit in memory by aligning the code with extra padding (when needed).

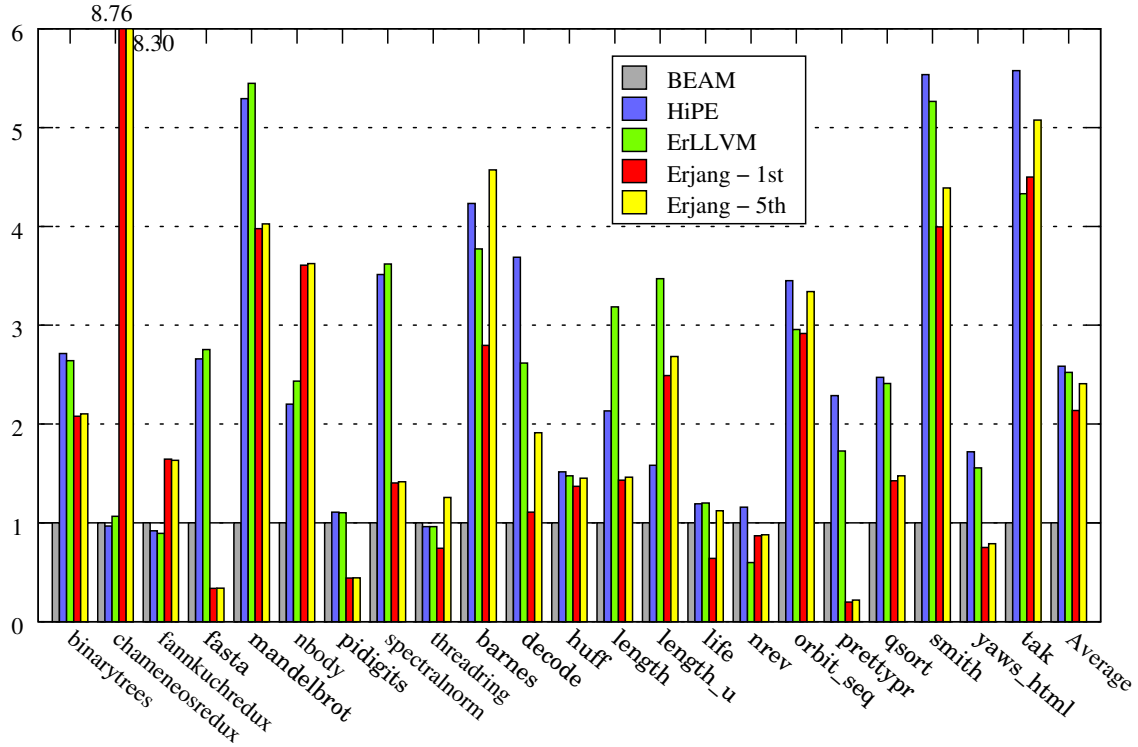

Figure 9: Speedups on x86_64 (normalized to BEAM).

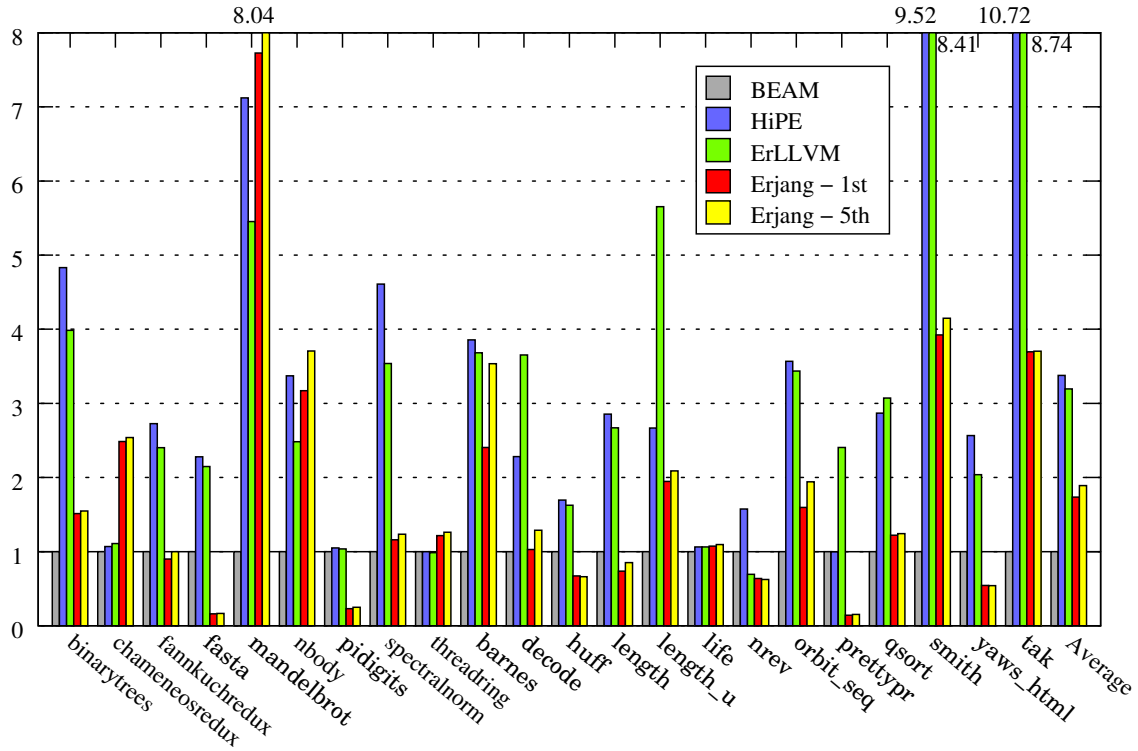

Figure 10: Speedups on x86 (normalized to BEAM).

	HiPE	ErLLVM	HiPE/ ErLLVM
Code Size	5504880	6625368	0.83
Compilation Time	427.29	547.89	0.78

(a) `x86`

	HiPE	ErLLVM	HiPE/ ErLLVM
Code Size	6607584	7915928	0.84
Compilation Time	497.64	541.70	0.92

(b) `x86_64`

Table 2: Binary code sizes (in bytes) and compilation times (in seconds) for `hipe` and `stdlib` applications.

In the current state of ErLLVM, reducing the size of the binary code has never been a top priority for us. However, we do believe that by the time we will have achieved better runtime performance (through more aggressive optimizations of the final code) the size of the binaries will have reduced too to the levels currently achievable by HiPE.

5.5 Compilation Times

We also compared the compilation times of ErLLVM when compiling the same two applications of the Erlang/OTP system on both supported architectures. Compilation times were obtained starting from the point where the BEAM bytecode is already present and is being disassembled (cf. Figure 1). Table 2 only displays the total results for both applications along with the ratios **HiPE/ErLLVM** of compilation times on x86_64 and x86.

We can see that the HiPE backend is approximately 10–20% faster than LLVM on average of 275 modules in both x86 and x86_64. This fact is not surprising since the LLVM backend is still immature, and attention was put to correctness rather than fast compilation. The main problem lies on the fact that the new backend uses an inefficient way for representing LLVM assembly and many intermediate files for passing data from one tool to the other. We tried to reduce the compilation time as much as possible by using buffered writes on files opened as `raw`[5]. Furthermore, for the translation to LLVM assembly, we use the final, unconverted SSA form of RTL provided by HiPE. Apparently, we go through all the optimizations that HiPE performs (described in Section 2.1) *plus* many optimizations that are provided by the LLVM optimizer and the LLVM code generator. So, many optimizations are performed more than once in different phases of the translation and this may not be worth it. We believe that we should use the simplest form of RTL (before being optimized at all) and let the LLVM infrastructure optimize it as much as possible. We consider compilation times as something that can be easily improved in the future.

5.6 Implementation Complexity

Finally, we evaluate the new LLVM backend in terms of implementation complexity. The term "complexity" refers to the amount of work required to build, maintain and extend the backend.

Each of the existing HiPE backends implements a compiler from RTL to symbolic target-specific assembly and an assembler to create binary code ready for loading into the runtime system. On the contrary, ErLLVM needs a (target-agnostic) translator from RTL to LLVM assembly and, in the current implementation, an object file parser to extract the generated information from the generated (on-disk) object file. It is clear for any compiler developer that mapping from one intermediate representation to another is by far easier than creating an assembler for each supported architecture.

[5] For more information regarding raw files consult the documentation of the `file` module in Erlang/OTP and `file:open/2` in particular.

HiPE Backend	Size (LOC)		
ARM	Total:	5352	
	Code:	3886	
	Blank:	636	
	Comments:	830	(17.6%)
SPARC	Total:	5137	
	Code:	3616	
	Blank:	643	
	Comments:	878	(19.5%)
x86/x86_64	Total:	10463	
	Code:	7424	
	Blank:	1056	
	Comments:	1983	(21.1%)
PPC/PPC64	Total:	6684	
	Code:	5001	
	Blank:	792	
	Comments:	891	(15.1%)
ErLLVM (x86/x86_64)	Total:	4824	
	Code:	3441	
	Blank:	439	
	Comments:	944	(21.5%)

Table 3: Code Sizes for various HiPE's backends. Measurements include only Erlang source (`.erl`) and header (`.hrl`) files; no Makefiles and other text files in the directories. The percentage of comments is estimated as: `Comments/(Code+Comments)`.

In Table 3 we present the sizes, in lines of codes (LOC), of the various HiPE backends. These numbers do not include common code that all backends share, such as code from the `flow`, `icode`, `rtl` and `misc` directories of the `hipe` application. It is clear that the LLVM backend is the simplest, both conceptually and in lines of code. In fact, only 2,563 lines of plain Erlang code correspond to the translator while 885 LOC belong to the `elf*_format` modules, that implement the object file parser.

6. Concluding Remarks and Future Work

This paper has described the architecture, design decisions and implementation details for a new backend for the native code compiler of Erlang/OTP that uses the LLVM compiler infrastructure for code generation. Special attention was paid from the beginning of the project to retain ABI compatibility with the other backends of the HiPE compiler and support all features of Erlang, such as hot-code loading, garbage collection, and exception handling. We evaluated the new backend with respect to two broad dimensions: performance and code complexity.

Our benchmark results indicate that the code generated from ErLLVM is significantly faster than BEAM, and on x86 and x86_64 achieves on average about the same performance as that of existing HiPE backends (Section 5.2). These results suggest that there are good indications that the LLVM backend, when extended to support all the architectures that HiPE currently supports, may become the default (only?) backend of the ahead-of-time native code compiler of Erlang/OTP. The main reason why this is likely to happen is that ErLLVM not only is a single code base that can offer the functionality of four different ones, but it is also easier to maintain and extend than all the other HiPE backends. This is a big gain and, taking into consideration that LLVM currently has a very active community of developers which is progressing quickly in all areas, HiPE can effortlessly benefit too, by outsourcing this work to LLVM.

Currently, the biggest drawbacks of ErLLVM are two: longer compilation times and the need for a custom version of LLVM. We are not surprised by the first one as, primarily, attention was paid to correctness and completeness rather than fast compilation. The compilation time can be improved significantly with the use of the LLVM bindings [8] in order to avoid printing to and parsing intermediate files. As far as the patches to LLVM are concerned, we have been in close contact with the LLVM developers in order to bring our changes to an acceptable form for the project. We expect to have them pushed upstream before the next stable release.

While the new backend is *complete* and rather *fast*, our work has clearly some way to go. Our primary focus in the future will be to improve ErLLVM's performance by properly taking advantage of many powerful features of LLVM that are currently underutilized, e.g., the Type-Based Alias Analysis (TBAA) [11] or the use of branch probabilities (that already exist in RTL) for better basic block placement. In the short run, our immediate goals involve: (a) changing the unit of compilation from function to module in order to enable intra-module optimizations, such as better inlining (while taking care of preserving hot-code loading semantics), and (b) using the LLVM bindings to avoid the overhead of intermediate files. In the long run, extending ErLLVM's translator to support all architectures currently supported by HiPE and ERTS (not just x86 and x86_64) would definitely be worthwhile.

Acknowledgments

We thank Katerina Roukounaki for suggestions that have improved the presentation of our work. We also thank Frits van Bommel, Duncan Sands, Samuel Crow and the other LLVM developers for their technical advice through the LLVMdev mailing list during the development of the LLVM backend. We particularly thank Jakob Stoklund Olesen for reviewing patches which are the foundation for including the extensions to LLVM that ErLLVM needs and for bringing us a step closer to integrating our work in the LLVM tree.

The ErLLVM project has been supported in part by Ericsson. Since November 2011 the research time of the first and the third author has been supported in part by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software".

References

[1] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.*, 16(3):428–455, May 1994.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[3] Free Software Foundation. The GNU C++ Library, 2011. URL http://gcc.gnu.org/onlinedocs/gcc-4.7.0/libstdc++/manual/manual/using_exceptions.html.

[4] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Virtual Execution Environment Conference (VEE 2010)*, Pittsburgh, USA, March 2010. ACM Press.

[5] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.

[6] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap architectures for concurrent languages using message passing. In D. Detlefs, editor, *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88–99, New York, NY, June 2002. ACM Press.

[7] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. *Springer International Journal of Software Tools for Technology Transfer*, 4(4): 421–436, Aug. 2003.

[8] L. Larsson. `llevm`: An Erlang wrapper to the C API functions of LLVM. URL http://www.github.com/garazdawi/llevm.

[9] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002. URL http://llvm.cs.uiuc.edu.

[10] LLVM Team. Exception Handling in LLVM, . URL http://llvm.org/docs/ExceptionHandling.html.

[11] LLVM Team. LLVM Language Reference Manual - TBAA metadata, . URL http://llvm.org/docs/LangRef.html#tbaa.

[12] D. Luna, M. Pettersson, and K. Sagonas. Efficiently compiling a functional language on AMD64: The HiPE experience. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 176–186, New York, NY, July 2005. ACM Press.

[13] MacRuby Project Team. MacRuby - Ruby for the Objective-C Runtime. URL http://macruby.org/.

[14] J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Trans. Prog. Lang. Syst.*, 26(4):735–765, July 2004.

[15] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, volume 2441 of *LNCS*, pages 228–244, Berlin, Germany, Sept. 2002. Springer.

[16] E. Phoenix and the Rubinius team. Rubinius - An implementation of the Ruby programming language. URL http://rubini.us.

[17] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.

[18] A. Rigo, M. Fijalkowski, C. F. Bolz, A. Cuni, B. Peterson, A. Gaynor, H. Ardo, H. Krekel, and S. Pedroni. The PyPy project. URL http://pypy.org.

[19] K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software – Practice and Experience*, 33(11):1003–1034, Sept. 2003.

[20] Shootout. The computer language benchmarks game, 2009. URL http://shootout.alioth.debian.org/.

[21] C. Stavrakakis and Y. Tsiouris. ErLLVM benchmarking infrastructure, 2012. URL https://github.com/chris-fren/erllvm-bench.

[22] D. A. Terei and M. M. T. Chakravarty. An LLVM backend for GHC. In *Proceedings of the Haskell Symposium*, pages 109–120, New York, NY, USA, Sept. 2010. ACM. doi: 10.1145/2088456.1863538. URL http://doi.acm.org/10.1145/2088456.1863538.

[23] K. K. Thorup. Erjang: A virtual machine for Erlang which runs on Java. http://github.com/trifork/erjang/wiki.

[24] Unladen Swallow community. Unladen Swallow - An optimization branch of CPython. URL http://code.google.com/p/unladen-swallow/.