HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang based on HiPE

Konstantinos Kallas[1] Konstantinos Sagonas[2]

[1]University of Pennsylvania
Philadelphia, USA

[2]Uppsala University
Uppsala, Sweden

Athens PL Seminar, 2018

Outline

Outline

Introduction

Erlang is a concurrent functional programming language designed for the development of:

- Scalable concurrent and distributed applications.
- Highly available and responsive systems.

Erlang/OTP, its main implementation, comes with:

- BEAM, a byte code compiler, which generates portable code.
- HiPE, a native code compiler, which generates efficient code.

Byte code and native code can coexist in the Erlang/OTP runtime system.

HiPErJiT

We present **HiPErJiT**, a profile-driven JiT compiler for Erlang based on HiPE.

In short, HiPErJiT:

- Preserves all the features and requirements of Erlang.
- Is maintainable and easy to keep in-sync with Erlang/OTP.
- Achieves decent performance.

Outline

Background
BEAM and HiPE

BEAM is the *de facto* virtual machine for Erlang.
It comes together with a byte code compiler that generates reasonably efficient and compact byte code, which is then executed by the VM.

HiPE is an ahead-of-time native code compiler for Erlang.
It improves the performance of Erlang applications by allowing users to compile their time-critical modules to native code.

Background
Challenges of compiling Erlang

Erlang has some characteristics that make generation of highly efficient code challenging:

- It is **dynamically typed**.
- It *requires* to be able to replace single modules of an application while the system is running (aka **hot code loading**).
- It makes a *semantic distinction* between module **local** and **remote calls**, which need to lookup the most recent version of a module.
- Its applications are **highly concurrent**, so a large number of processes may execute code from different modules at the same time.

Outline

Architecture

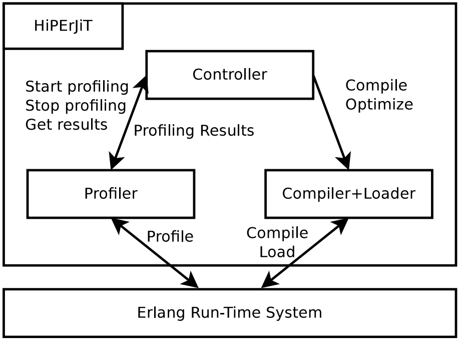

Figure 1: High level architecture of HiPErJiT.

Outline

Controller
Compilation condition

The controller decides which modules to compile and optimize based on the following condition:

$$FutureExecTime_c + CompTime < FutureExecTime_i$$

Controller

A more realistic compilation condition

The following assumptions are made:

- $FutureExecTime = ExecTime$
- $ExecTime_c * Speedup_c = ExecTime_i$
- $CompTime = C * Size$

The updated compilation triggering condition is:

$$\frac{ExecTime_i}{Speedup_c} + C * Size < ExecTime_i$$

Controller

Estimating compilation time C

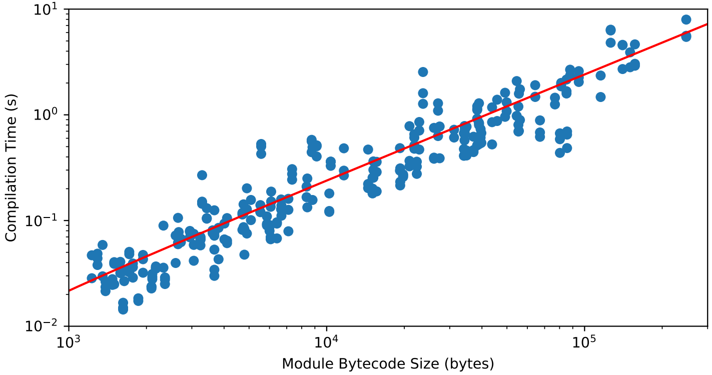

Figure 2: Compilation times of modules related to their bytecode size.

Outline

Profiler

Architecture

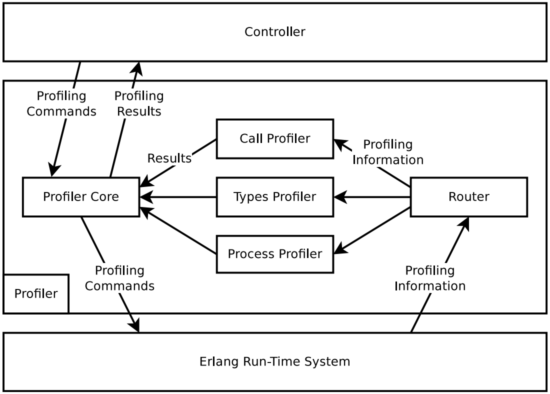

Figure 3: Profiler architecture and internal components.

Profiler

Profiling Execution Time

The call profiler receives messages of the form:

$$\langle Action, Timestamp, Process, Function \rangle$$

where $Action = \text{call} \mid \text{return\_to} \mid \text{sched\_in} \mid \text{sched\_out}$.

For each sequence:

$$[\langle \text{sched\_in}, T_1, P, F_1 \rangle, \langle A_2, T_2, P, F_2 \rangle, \ldots, \langle \text{sched\_out}, T_n, P, F_n \rangle]$$

the time spend at each function F_i can be computed by:

$$[\langle T_2 - T_1, F_1 \rangle, \langle T_3 - T_2, F_2 \rangle, ..., \langle T_n - T_{n-1}, F_{n-1} \rangle]$$

Profiler
Profiling Argument Types (i)

Arguments of functions could be arbitrarily large, and determining their type requires a complete traversal. The profiler instead approximates their types using the *depth-k abstraction*, a limited-depth term traversal.

For example, the term:

```
{foo,{bar,[{a,17},{a,42}]}}
```

has type:

```
tuple2('foo',tuple2('bar',list(tuple2('a',17|42))))
```

Its depth-k type abstractions are:

- depth-1: `tuple2('foo',tuple2(any(),any()))`
- depth-2: `tuple2('foo',tuple2('bar',list(any())]])`

Profiler
Profiling Argument Types (ii)

Erlang collections (lists, maps, etc.) can contain elements of different types. For example, `[1, b, 3.14, {d,42}]` is a valid list. Finding their complete type needs a traversal.

However, in practice most collections contain elements of the same type. Therefore, HiPErJiT optimistically estimates their types by only considering a subset of their elements.

Profiler
Probabilistic Profiling

Profiling heavily concurrent applications leads to overhead as the profiler receives a lot more messages than it can handle, and becomes a bottleneck.

HiPErJiT employs probabilistic profiling by maintaining a tree of all alive processes. The nodes of the process tree are of the following type:

```
-type ptnode() :: {pid(), mfa(), list(ptnode())}.
```

Instead of profiling all the leaf processes, HiPErJiT profiles a random subset of leaf processes that have been spawned using the same MFA.

Outline

Compiler+Loader

This component receives the collected profiling data, calls a version of the HiPE compiler extended with two additional optimizations, and loads the JiT compiled module.

The major challenge is that ERTS supports hot code loading and only allows up to two versions of the same module to be simultaneously loaded. If a user reloades a module while the system is running, HiPErJiT automatically retriggers the process of reprofiling this module.

Compiler+Loader
Loading Race Condition

If the user loads a new version of a module after HiPErJiT has started compiling the current one, but before it has completed loading it, the JiT-compiled code for an older version of the module will be loaded.

Thus, HiPErJiT loads a module *m* as follows:

1. Acquire a module lock, not allowing any other process to reload module *m*.
2. Call HiPE to compile *m* to native code.
3. Repeatedly check whether any process executes m_{old} code. When no process executes m_{old} code, load the JiT-compiled version of *m*.
4. Release the lock so that new versions of *m* can be loaded again.

Outline

Outline

Type Specialization

In dynamic languages, there is typically very little type information available during compilation. Therefore, dynamic language compilers generate less efficient code.

- The generated code handles all possible value types, by introducing type tests to ensure that operations are performed on terms of the correct type.
- All values are tagged, which means that their runtime representation contains information about their type.

HiPErJiT tackles this problem with a combination of **optimistic type compilation** and **static type analysis**.

Optimistic Type Compilation

Creates specialized function versions, whose arguments are of known types, based on the type information that has been collected by profiling.

For each function `f` where the collected type information is non-trivial:

1. `f` is duplicated into `f$opt` and `f$std`.
2. A header function that contains all the type tests is created. If all tests pass, `f$opt` is called, otherwise `f$std`.
3. The header function is inlined in every local function call of `f`.

Type Analysis

Type analysis is an optimization pass that infers type information for each program point and simplifies the code based on this information.

It removes unnecessary:

- Type tests and checks
- Boxing and unboxing floating-point operations

Outline

Inlining

Inlining improves performance in two ways:

- It mitigates the function call overhead
- It enables more optimizations to be performed later

However, aggressive inlining has several potential drawbacks, both in compilation time, as well as in code size increase.

In order to get performance benefit from inlining, the compiler must achieve a fine balance between inlining every function call and not inlining anything at all.

Inlining
Inlining decision

HiPErJiT makes its inlining decision based on run-time information combining the ideas of call frequency and call graphs.

It iteratively inlines all the local calls to F_2 in F_1 if the pair (F_1, F_2) has the highest call frequency.

It restrains inlining by setting a limit on how much each module can grow. Small modules are allowed to grow up to double their size, while larger ones are only to grow by 10% of their current size.

Outline

Evaluation

We evaluate the performance of HiPErJiT against BEAM, HiPE, and Pyrlang.

All experiments were conducted on a laptop with an Intel Core i7-4710HQ @ 2.50 GHz CPU and 16 GBs of RAM running Ubuntu 16.04.

Evaluation
Profiling Overhead

As a worst-case estimate we measured the profiling overhead of HiPErJiT.

We executed around 50 benchmarks with a modified version of HiPErJiT that does not compile any module, and the average slowdown was measured to be 10%, ranging from 5% in most cases, up to 40% in a heavily concurrent program.

We also separately measured the benefit of probabilistic profiling over standard profiling on around 10 concurrent benchmarks.
The average slowdown from standard profiling was about 19%, while the average slowdown from probabilistic profiling was about 13%.

Evaluation

Average speedup over BEAM for the small benchmarks

Configuration	Speedup
HiPE	2.08
HiPE + Static inlining	2.17
Pyrlang	1.05
HiPErJiT	1.85
HiPErJiT w/o 1st run	2.08
HiPErJiT last 50% runs	2.33

Figure 4: Speedup over BEAM for the small benchmarks.

Evaluation

Speedup over BEAM on small benchmarks (i)

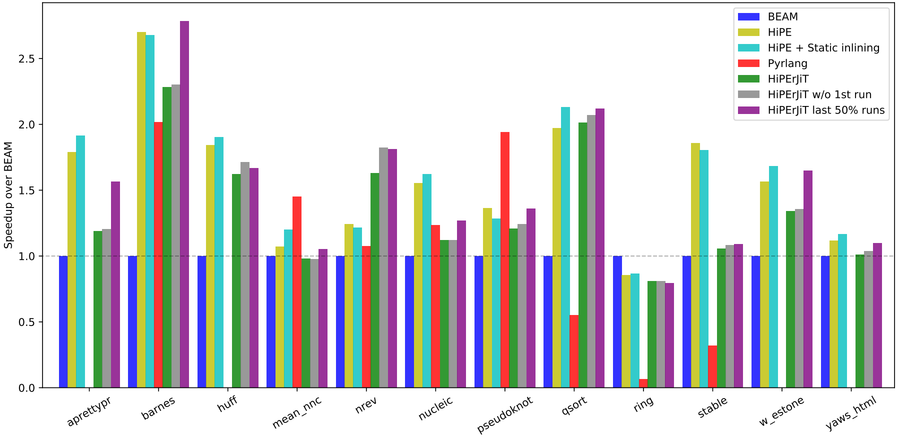

Figure 5: Speedup over BEAM on small benchmarks.

Evaluation

Speedup over BEAM on small benchmarks (ii)

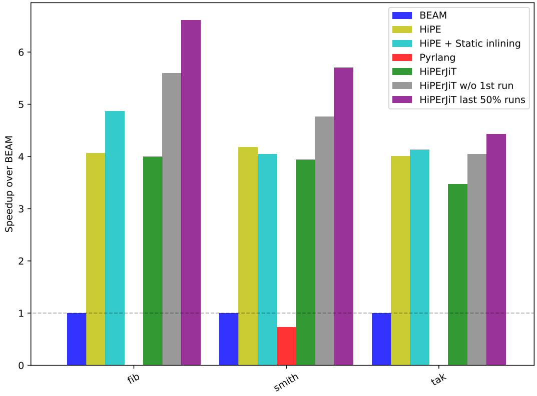

Figure 6: Speedup over BEAM on small benchmarks.

Evaluation

Average speedup over BEAM for the CLBG benchmarks

Configuration	Speedup
HiPE	2.05
HiPE + Static inlining	1.93
HiPErJiT	1.78
HiPErJiT w/o 1st run	2.14
HiPErJiT last 50% runs	2.26

Figure 7: Speedup over BEAM for programs from the CLBG.

Evaluation on a bigger program

	Dialyzer Speedup	
Configuration	Building PLT	Analyzing
HiPE	1.73	1.70
HiPE + Static inlining	1.75	1.72
HiPErJiT	1.51	1.30
HiPErJiT w/o 1st run	1.58	1.35
HiPErJiT last 50% runs	1.62	1.37

Figure 8: Speedups over BEAM for two Dialyzer use cases.

Outline

Conclusion

We have presented HiPErJiT, a profile-driven Just-in-Time compiler based on HiPE.

- It offers performance which is better than BEAM's and comparable to HiPE's.
- It preserves all important features for Erlang's application domain.
- For maintainability, it utilizes some existing Erlang/OTP components.
- It adds some new components that piggyback on existing infrastructure of the HiPE compiler.

Future Work

- Investigating techniques that reduce the profiling overhead of HiPErJiT, especially in heavily concurrent applications.
- Evaluate HiPErJiT on long-running real world applications, as they are the ideal focus for continuous run-time optimization.
- Improve the stability and precision of HiPErJiT 's profiling mechanisms.
- Introduce more effective type optimizations that could benefit from type specialization and inlining.
- Improve the performance of message passing using profiling data.

Future Work

- Investigating techniques that reduce the profiling overhead of HiPErJiT, especially in heavily concurrent applications.
- Evaluate HiPErJiT on long-running real world applications, as they are the ideal focus for continuous run-time optimization.
- Improve the stability and precision of HiPErJiT 's profiling mechanisms.
- Introduce more effective type optimizations that could benefit from type specialization and inlining.
- Improve the performance of message passing using profiling data.

Questions?