All you wanted to know about the HiPE compiler (but might have been afraid to ask)

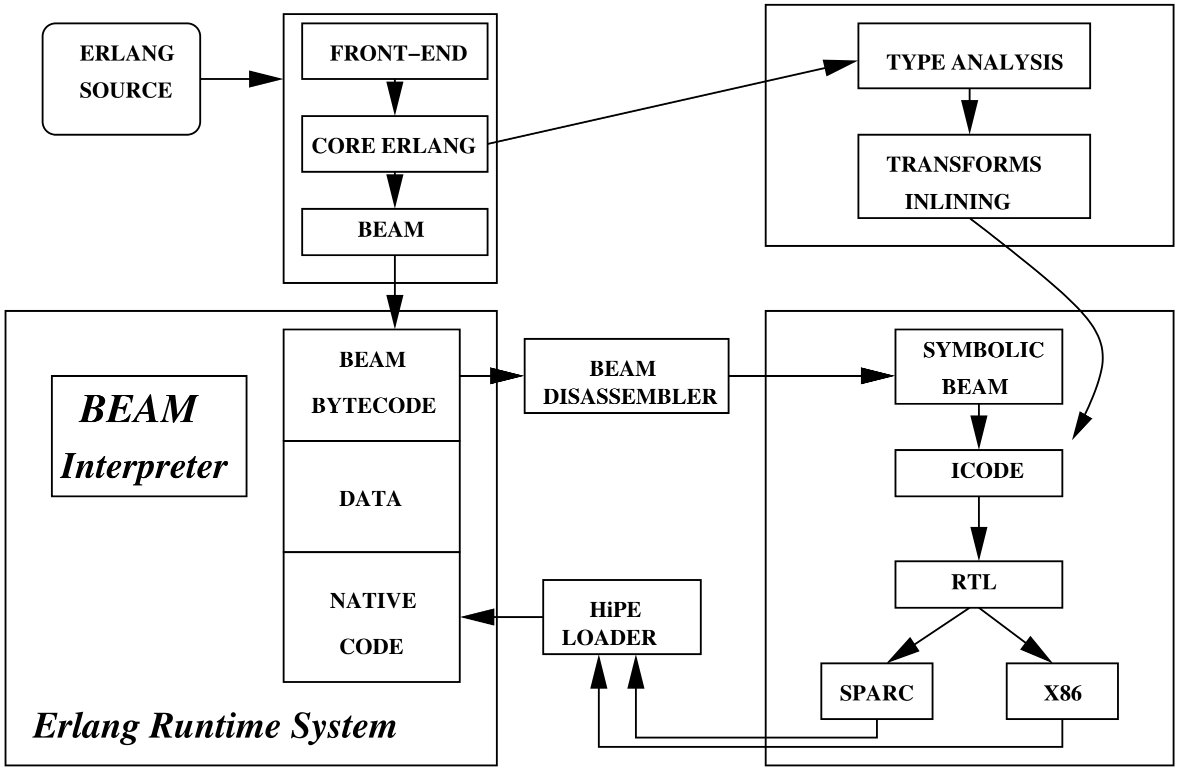
K. Sagonas, M. Pettersson, R. Carlsson,
P. Gustafsson, T. Lindahl

The High-Performance Erlang Group
Computing Science Department
Uppsala University, Sweden

Contents

- Brief overview of HiPE's architecture.
- HiPE–HOWTO: compiling and loading code, compiler options
- Recent improvements.
- Planned extensions.
- HiPE do's and dont's.
- (The few) Known incompatibilities and limitations.

HiPE's architecture



HiPE–HOWTO

Simply add the `native` flag to the usual BEAM compilation tools!

```
1> c(Module, [native]).
```

compile and load interactively

```
2> compile:file(Module, [native]).
```

compile to `.beam` file with both BEAM and native code; native code in a `.beam` file is automatically enabled at load-time

```
erlc +native Module.erl
```

in a Unix shell or Makefile

HiPE–HOWTO, continued

For HiPE-specific options, add `{hipe,Optionlist}` to the list of options (in addition to `native`):

```
c(Module, [native, {hipe, [o3]}]).
erlc +native '+{hipe, [o3]}' Module.erl
-compile({hipe, [o3]}).
```

- `o0`, `o1`, `o2`, `o3` Choose optimisation level. `o2` is default.
- `{regalloc,linear_scan}` Specify the use of a specific register allocator. `coalescing` is default on x86 (and SPARC at `o3`), but it can be slow on large functions.

HiPE–HOWTO, continued

To find out what options there are and what they do, use `hipe:help()`, `hipe:help_options()`, and `hipe:help_option(Option)`. For example:

`hipe:help_option(regalloc)` tells you what register allocators you can choose from.

`hipe:help_option(o3)` tells you what internal compiler options `o3` enables.

Recent improvements

Bit syntax

- In R9B, HiPE recognised special-cases of BEAM's bit-syntax instructions, and implemented them with native code. Other cases became calls to C functions. Speedup over BEAM generally between 1.0–2.0.
- In R9C, HiPE identifies the entire block of BEAM instructions for a binary matching or creation expression, and compiles it as a single unit. This results in better code and fewer calls to C functions. Speedup over BEAM in the 1.5–4.0 range.

Bit syntax speedups in R9C

Benchmark	speedup SPARC	speedup x86
bsextract	3.30	4.13
bsdecode	1.23	1.76
ber_decode	2.44	2.57
bsencode	1.53	2.30
ber_encode	1.91	2.08

Recent improvements

Floating-point arithmetic

- A single f.p. operation becomes: type test, move value from heap to FPU, do operation, allocate heap, move value from FPU to heap, tag pointer.
- Starting with R9B, BEAM recognises blocks of f.p. operations, and uses new instructions for unboxed f.p. arithmetic. BEAM now skips the allocate, store, tag, test, load sequence between pairs of f.p. operations.
- HiPE goes further and keeps intermediate f.p. values in the FPU rather than in memory "registers". This reduces memory traffic and delays between f.p. instructions.

Floating-point benchmark speedups

Benchmark	speedup SPARC	speedup x86
float_bm	3.66	2.57
barnes-hut	2.39	2.51
fft	1.88	1.95
wings	1.12	1.36
raytracer	1.07	1.12
pseudoknot	2.16	2.71

Recent improvements

Local type analysis

- Propagates type information from guards and other operations that yield known types.
- Works on a single function at a time.
- Eliminate redundant type tests.
- Specialise overloaded operations (e.g. `+`) when types are known.
- No performance results yet, but code size is reduced, which also makes remaining compiler passes faster.

Recent improvements

Miscellaneous

- Apply (`M:F(...)` calls) is now implemented natively. Major speedup in apply-intensive code. Also improves "generic server".
- Mailbox operations in receives are now inlined. 10–15% improvement in receive/timer-intensive benchmarks.
- Compile-time literals are merged, reducing their space usage.
- A source of excessive compile-times has been fixed.

Planned extensions

Module-global type analysis

Extends the local type analysis with information about actual parameters and return values from function calls.

- More opportunities for type check removal and specialising overloaded operations.
- Tuples can be returned in registers or on the stack.

The precision of the analysis depends the amount of code it sees at a time, the size of the external interfaces, and on programming style.

Planned extensions

Module-global compilation from source

Instead of compiling a function at a time, via its BEAM code, we are working on compiling from Erlang source, via Core Erlang.

- Type analysis is potentially more precise when done at a level closer to normal Erlang.
- Easier to utilise high-level meta-data (e.g. types) for things like function inlining and optimised calling conventions.
- Avoiding BEAM code gives us better control over constructs like pattern-matching and bit syntax operations.

Do's and Dont's

- Basic fact #1: Compiling Erlang code to native code makes *it* run faster, but doesn't change anything else.
- Basic fact #2: Switching modes (from native to BEAM and vice versa) is more expensive than a simple function call.

Do's and Dont's, continued

Do compile your code if it performs a lot of computations.

- Traditional functional code with lots of function calls, data construction, and pattern matching, clearly benefits from being compiled to native code.

  Deep recursions are several times faster in native code due to explicit garbage collection support for deep native stacks.

- Bit-syntax operations on binaries may be several times faster in native code.

- Blocks of floating-point operations may be several times faster in native code.

Do's and Dont's, continued

Helping the compiler:

- Bit-syntax optimisations rely on segments being 8-bit aligned. Variable-length segments should use a multiple-of-8 unit.

  32-bit floats in binaries aren't handled efficiently.

- Monomorphic code is more likely to be translated to good native code. Using guards, e.g., `is_float/1`, may help the compiler detect and utilise relevant type restrictions.

- Order clauses so that more likely cases precede less likely cases.

Do's and Dont's, continued

Helping the compiler, continued:

- Largish self-contained modules with narrow external interfaces help type analysis and inlining decisions.

  The `-compile(export_all)` directive makes the external interface wide-open, which limits the precision of type analysis and makes inlining less likely to occur.

  At a remote call, the compile cannot in general make any assumptions about the called function or what it will return.

Do's and Dont's, continued

While in native code, avoid leaving it.

- Avoid frequent calls to BEAM-mode functions.
- Ensure that Erlang libraries you use frequently are compiled to native code. This includes standard libraries.
- If you call function closures in native code, ensure that they also come from native code.
- It's best to first profile the application, and then compile the most frequently used parts to native code.

Do’s and Dont’s, continued

Don’t compile your code if time is mostly spent elsewhere.

- Some applications are “BIF bound”, being mostly glue around things like ETS tables or drivers.
- Some applications are “scheduling bound”, being mostly send and receive agents that don’t do much actual computation.

Compiling these can only yield limited performance improvements.

Known incompatibilities

- Stack dumps at exceptions are less detailed than in BEAM. Assume `bug(X) -> X+1.` Then calling `bug(a)` generates

  `{badarith,[]}`

  if `bug/1` is in native code, but

  `{badarith,[{Module,bug,1},...]}`

  if `bug/1` is in BEAM code.

- `Fun = {M,F}, Fun()` is not supported. Use proper `funs` or `M:F()` calls instead.

- Floating-point on x86 uses higher precision for intermediate values and calculations than BEAM does, so HiPE may deliver slightly different (more accurate) results.

Current limitations

- Native code is never freed. Repeatedly loading newer versions of the code leads to a (bounded) space leak.

  Fixing this requires a redesign of the HiPE object code loader.

- There is a *single* global area for compile-time constant terms, and this area may overflow if large amounts of native code is loaded.

  This is a consequence of the design of Erlang/OTP's memory management system. We are hoping that R10 will fix this.