HiPE 1.0.2
User's Manual
BETA Version 0.9

Erik Johansson

January 25, 2002

Contents

1 Introduction

The HiPE Erlang system version 1.0.2 is based on the BEAM version of Open Source Erlang[1] version R8.

1.1 About this document

This documents describes how to use the HiPE native code compiler. It uses the same conventions for describing ERLANG modules as the "OTP Reference Manual".

This document does not describe ERLANG or the OTP ERLANG system. For more information about ERLANG and Open Source ERLANG please refer to:
`http://www.erlang.org/`
and for more information about OTP and Erlang Systems refer to:
`http://www.erlang.se/`.

2 Getting started

2.1 Getting HiPE

You get HiPE by downloading the latest version of Open Source Erlang from `http://www.erlang.org/`.[2] HiPE is available for SPARC/Solaris and x86/Linux (x86/Solaris) at the moment. (It might work on x86/Win32.)

This will get you a gziped tar archive, unzip and untar it in some directory (which we now will call `$HIPEROOT`).

```
>tar xzf otp_src_R8B-0.tar.gz
>export $HIPEROOT=otp
```

2.2 Building HiPE

Assuming that you have the source code for the HiPE system in the directory `$HIPEROOT`:

```
> cd $HIPEROOT
> ./configure --enable-hipe
[...]
> make
[...]
```

If you want to use the unified heap architecture you should also give the flag `--enable-unified-heap` to configure.

[1] `http://www.erlang.org/`
[2] The current version is at http://www.erlang.org/download/otp_src_R8B-0.tar.gz

2.3 Starting HiPE

Assuming that you have built HiPE in the directory `$HIPEROOT`, you can start HiPE with:
`$HIPEROOT/bin/erl`

3 Compiling a module

With the HiPE system you can compile a module to native code just as you would compile it to beam code by just giving the `native` flag to the compiler. E.g. From the shell:
```
> $HIPEROOT/bin/erlc +native fop.erl
```
From the erlang shell:
```
1> c(foo,[native]).
```

You can supply a list of compilation options to the compiler (see Section 4.2) as follows, to erlc:
```
> erlc +native  +\{hipe,\[option1,option2\]\} foo.erl
```
Note that brackets used in ERLANG terms usually have to be quoted when supplied from the shell (to get the right number of quotes in a script is usually quite (tr)icky). In the erlang shell you can write these options somewhat more clearly:
```
1> c(foo,[native, {hipe,[option1,option2]}]).
```

From the ERLANG shell and in ERLANG programs you can also use the `hipe` module to, on the fly, compile functions and modules to native code (see Section 4).

4 Module hipe

This module contains functions for compiling to native code.

4.1 Exported Functions

`c(Source) -> Res`

Types:

Source = Module — {Module,Function,Arity}

Module = atom()

Function = atom()

Arity = integer()

Res = {ok,Module}

Compiles the module Module (or the function Module:Function/Arity) to native code and links the native code into the system. A BEAM compiled version of the module must be accessible through the

ERLANG path or already loaded. If the module does not exist an exception is thrown.

`c(Source,Options) -> Res`

Types:

Source = Module — {Module,Function,Arity}

Module = atom()

Function = atom()

Arity = integer()

Options = list()

Res = {ok,Module}

Compiles the module Module to native code with the compiler options given by Options, and links the native code into the system. A BEAM compiled version of the module must be accessible through the ERLANG path or already loaded.

Available compiler options are listed in the *Options* section below.

`help() -> ok`

Writes a short description of available exported functions in hipe to stdout.

`version() -> Version`

Types:

Version = {Major, Minor, Increment}

Major = Minor = Increment = integer()

Returns the current version number of HiPE as a tuple.

4.2 Compiler Options

! → **NOTE: This section is incomplete and a bit inaccurate at the moment, use `hipe:help_options()` and `hipe:help_option(foo)` for up to date information (or look in the source).**

The argument `Options` is a list of options (for example `[o2, pp_sparc]`).

`o1` : Some optimizations.

`o2` : Several optimizations.

`icode_rename` : Rename variables.

`icode_prop` : One pass of Icode optimization. Semi-local copy propagation, constant propagation, constant folding and dead code removal. Works on extended basic blocks. No iteration is done at this point.

`icode_bwd_cprop` : Backward copy propagation on Icode.

`rtl_prop_1` : First pass of RTL optimization. Semi-local copy propagation, constant propagation, constant folding and dead code removal. Works on extended basic blocks. No iteration is done at this point.

`rtl_prop_2` : second pass of RTL optimization, done after gc-test expansion, save and restore added, frame expansion.

`sparc_schedule` : Perform ILP scheduling on code (also annotates code with cycle estimates).

`!` → **NOTE: This optimization might produce incorrect results.**

`sparc_post_schedule` : Schedule after register allocation as well.

`!` → **NOTE: This optimization might produce incorrect results.**

`pp_all` : Pretty Print All, expands to:
`[pp_beam, pp_icode, pp_rtl, pp_sparc]` or
`[pp_beam, pp_icode, pp_rtl, pp_x86]` depending on target architecture.

`pp_X` : Pretty prints code of type X where X is one of `beam`, `icode`, `rtl`, `sparc`, or `x86`.

`{pp_X,{only, Lst}}` : Pretty prints code of type X only if the function (MFA) is a member of the list `Lst`.

`{pp_X,{file, FileName}}` : Pretty prints code of type X to the file `FileName`.

`verbose` : Print some progress reports to stdout.

`time` : Print timing information for the different stages of the compilation.

You can also write `{no,Option}` to turn the `Option` off. In that case, it will look to the system as if you haven't specified the option (and it won't be used as a default option either)

`!` → NOTE: the options are processed in the order they appear; an early option will 'shadow' a later one.

Example: `has_option([cse,{no,cse}]) = true`

5 Profiling

To find the hot-spots of a program you can use the hipe-bifs for call counts. These work on emulated code by patching the first

instruction in each function to a special hipe/BEAM instruction which increments a counter on each call.

I have provided a small profiling module below (hipe_profile), to make life easier. You can call the profile function in this module with a closure of arity 1 and the argument to apply, and you will get back a list of all loaded functions and the number of times they are called.

E.g.:

```
> hipe_profile:profile(fun (A) -> fib:test(A) end,{20,1}).
[{{fib,fib,1},21891},
 {{lists,map,2},3241},
 {{lists_sort,keysplit_1,8},2269},
 {{hipe_profile,'-res_module/1-fun-0-',2},2200},
 ...
```

Note though that this profiling is slightly contaminated with calls executed by the profiler. If you try it out on a larger program the profiler will just be background noise...

```
2> List = hipe_profile:profile(fun (A) ->
                                   eddie:test(A)
                end,700).
[{{regexp,re_apply,4},2212000},
 {{http_parse,find_two_newlines,2},2198000},
 {{http_parse,split_list,3},1855000},
 {{srv_parse,request_uri,2},588000},
 {{srv_parse,field_value,2},525000},
 {{srv_parse,field_name,2},392000},
 {{regexp,re_apply_more,3},364000},
 {{regexp,in_char_class,2},287000},
 {{lists,reverse,1},245722},
 {{regexp,first_match,3},245000},
 {{regexp,re_apply,3},231000},
 {{string,substr,2},224000},
 {{srv_parse,tolower,1},210000},
 {{string,substr1,2},203000},
 {{srv_parse,key,2},196000},
 {{regexp,reg3p,2},189000},
 {{regexp,reg2p,2},168000},
 {{regexp,reg3,1},168000},
 {{regexp,reg4,1},168000},
 {{regexp,re_apply_or,2},147000},
 {{http_parse,first_part,2},126000},
 {{lists,append,1},126000},
 {{lists,any,2},119000},
 {{http_fields,get_header_value,2},106400},
```

```
 {{lists,flatmap,2},105000},
 {{srv_parse,skip_lwsp|...},91000},
 {{lists|...},84000},
 {{...}|...},
 {...}|...]
3>
```

This list can then be used to selectively compile functions to native code. E.g. compiling functions called more than 100.000 times:

```
> Comp = fun(ProfList,Threshold) ->
           [hipe:c(F) ||
              {F,Count} <- ProfList,
              Count > Threshold]
         end.
...
> Comp(List,100000).
```

This module will probably be integrated into the HiPE system someday, but until then you get the source code here. Code for hipe_profile:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Copyright (c) 2001 by Erik Johansson.
%% ===================================================
%%  Filename :  hipe_profile.erl
%%  Module   : hipe_profile
%%  History  : * 2001-07-12 Erik J (happi@csd.uu.se):
%%               Created.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-module(hipe_profile).
-export([profile/2,
 prof/0,prof_off/0,clear/0,res/0,
 prof_module/1, prof_module_off/1,
clear_module/1, res_module/1]).

profile(F,A) ->
  F(A),
  prof(),
  clear(),
  F(A),
  R=res(),
  prof_off(),
  R.

prof() ->
  mods(fun prof_module/1),ok.
prof_off() ->
  mods(fun prof_module_off/1),ok.
clear()->
```

```
  mods(fun clear_module/1),ok.
res() ->
  lists:reverse(
    lists:keysort(2,
      lists:flatten(
        mods(fun res_module/1)))).

mods(F) ->
  [F(X) || X <- mods()].

mods() ->
  [element(1,X) || X<- code:all_loaded()].

prof_module(Mod) ->
  Funs = Mod:module_info(functions),
  lists:map(
    fun ({F,A}) ->
      catch hipe_bifs:call_count_on({Mod,F,A})
    end,
    Funs),
  ok.

prof_module_off(Mod) ->
  Funs = Mod:module_info(functions),
  lists:map(
    fun ({F,A}) ->
      catch hipe_bifs:call_count_off({Mod,F,A})
    end,
    Funs),
  ok.

clear_module(Mod) ->
  Funs = Mod:module_info(functions),
  lists:map(
    fun ({F,A}) ->
      catch hipe_bifs:call_count_clear({Mod,F,A})
    end,
    Funs),
  ok.

res_module(Mod) ->
  Funs = Mod:module_info(functions),
  lists:reverse(lists:keysort(2,lists:map(
    fun ({F,A}) ->
      MFA = {Mod,F,A},
      {MFA,
       case catch hipe_bifs:call_count_get(MFA) of
         N when integer(N) -> N;
         _ -> 0 end
```

```
  }
end,
Funs))).
```