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
Information Technology Department, Uppsala University, Sweden
hipe@csd.uu.se

ABSTRACT

We present a user-oriented description of features and characteristics of the High Performance Erlang (HiPE) native code compiler, which nowadays is part of Erlang/OTP. In particular, we describe components and recent additions to the compiler that improve its performance and extend its functionality. In addition, we attempt to give some recommendations on how users can get the best out of HiPE's performance.

1. INTRODUCTION

During the last few years, we have been developing HiPE, a high-performance native code compiler for Erlang. HiPE offers flexible, fine-grained integration between interpreted and native code, and efficiently supports features crucial for Erlang's application domain such as light-weight concurrency. HiPE exists as a new component (currently about 80,000 lines of Erlang code and 15,000 lines of C and assembly code) which nowadays is fully integrated with Ericsson's Erlang/OTP implementation; in fact, HiPE is available by default in the open-source version of R9. The HiPE compiler currently has back-ends for UltraSPARC machines running Solaris and Intel x86 machines running Linux or Solaris.

The architecture and design decisions of HiPE's SPARC and x86 back-ends have been previously described in [5] and [11] respectively. A brief history of HiPE's development appears in [6]. As performance evaluations in these reports show, HiPE considerably improves the performance characteristics of Erlang programs, and on small sequential programs makes Erlang/OTP competitive in speed to implementations of other 'similar' functional languages such as Bigloo Scheme [13] or CML (Concurrent SML/NJ [12]).

Performance evaluation aside, all the above mentioned reports address quite technical compiler and runtime system implementation issues which most probably are not so informative for Erlang programmers who are simply interested in using HiPE for their everyday application development.

To ameliorate this situation, the current paper is targeted towards HiPE users. Its aims are to:

1. describe features – and sometimes secrets – of the HiPE compiler that are of interest to its users;
2. introduce recent and planned additions to the HiPE compiler in a way that focuses on how these new features affect users (i.e., without obfuscating their presentation by getting deep into technical details); and
3. give recommendations on how users can get the best out of HiPE's performance.

To make the paper relatively self-contained and provide sufficient context for the rest of its contents, Section 2 begins by overviewing HiPE's current architecture, then describes basic usage, compiler options and recent improvements, and finally presents some extensions to HiPE's functionality which are currently underway and will most probably be included in release R9C. Section 3 offers advise on HiPE's use, followed by Section 4 which reveals and documents limitations and the few incompatibilities that currently exist between the BEAM and the HiPE compiler. Finally, Section 5 briefly wraps up.

We warn the reader that the nature of certain items described in this paper is volatile. Some of them are destined to change; hopefully for the better. HiPE's homepage[1] might contain a more up-to-date version of this document.

2. HIPE COMPILER: A USER-ORIENTED OVERVIEW

2.1 HiPE's architecture

The overall structure of the HiPE system is shown in Fig. 1. The Erlang/OTP compiler first performs macro preprocessing, parsing, and some de-sugaring (e.g., expanding uses of the record syntax) of the Erlang source code. After that, the code is rewritten into Core Erlang [2, 1]. Various optimizations such as constant folding, and (optional) function inlining, are performed on the Core Erlang level. After this, the code is again rewritten into BEAM virtual machine code, and some further optimizations are done. (The BEAM is the de facto standard virtual machine for Erlang, developed by Erlang/OTP. It is a very efficiently implemented register machine, vaguely reminiscent of the WAM [14].)

The HiPE compiler has traditionally started from the BEAM virtual machine code generated by the Erlang/OTP

[1] http://www.csd.uu.se/projects/hipe/

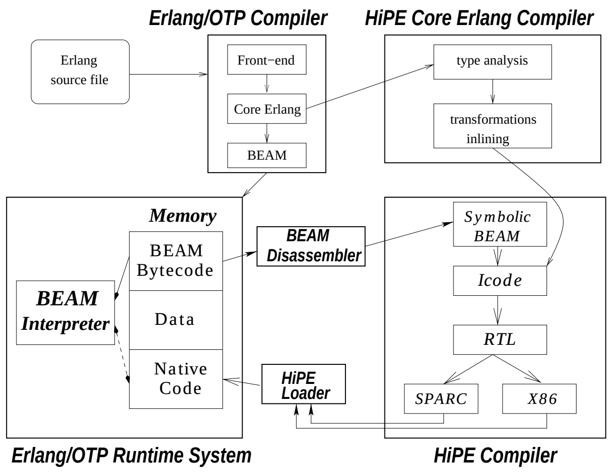

Figure 1: Structure of a HiPE-enabled Erlang/OTP system.

compiler. The BEAM code for a single function is first translated to ICode, an assembly-like language with a high-level functional semantics. After optimizations, the ICode is translated to RTL ("register-transfer language"), a low-level RISC-like assembly language. It is during this translation that most Erlang operations are translated to machine-level operations. After optimizations, the RTL code is translated by the backend to actual machine code. It is during this translation that the many temporary variables used in the RTL code are mapped to the hardware registers and the runtime stack. Finally, the code is loaded into the runtime system.

The Erlang/OTP runtime system has been extended to associate native code with functions and closures. At any given point, a process is executing either in BEAM code or in native code: we call this the *mode* of the process. A *mode switch* occurs whenever control transfers from code in one mode to code in the other mode, for instance when a BEAM-code function calls a native-code function, or when the native-code function returns to its BEAM-code caller. The runtime system handles this transparently, so it is not visible to users, except that the native code generally executes faster.

A new feature, described further below, is that the HiPE compiler can compile directly from Core Erlang. When used in this way, the compiler compiles a whole module at a time, and performs global analyses and optimizations which are significantly more difficult to perform (and thus not available) in the traditional mode.

2.2 Basic usage

The normal way of using the HiPE native code compiler is via the ordinary ERLANG compiler interface, by adding the single compilation option `native`. From the ERLANG shell, using the `c` shell function, this looks as follows:

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

This will compile the file `my_module.erl` to native code and load the code into memory, following the normal module versioning semantics of ERLANG.

Calling the standard compiler function `compile:file/2` (which by default does not load the resulting code) will produce a `.beam` file that contains both the native code *and* the normal BEAM code for the compiled module; e.g.:

```
compile:file(my_module, [native])
```

produces a file `my_module.beam` which can be loaded later. When a `.beam` file is loaded, the loader will first attempt to load native code, if the file contains native code that is suitable for the local system, and only if this fails is the BEAM code loaded. In other words, the `.beam` files may be "fat", containing code for any number of different target machines.

The compiler can also be called from the external program `erlc` (which indirectly calls the `compile:file/2` function). E.g., from a UNIX command line shell or make-file:

```
erlc +native my_module.erl
```

producing a file `my_module.beam`.

Additional compiler options may be given between the `erlc` command and the file name by prefixing them with `+`. Quoting may be necessary to avoid expansion by the shell, as for example in:

```
erlc +native +'{hipe,[verbose]}' my_module.erl
```

Generating native code and loading it on-the-fly into the system is possible even in cases when the ERLANG source code is not available but the `.beam` file (containing BEAM bytecode) exists. This can be done for whole modules using:

```
hipe:c(my_module)
```

or even for individual functions {`M,F,A`} using:

```
hipe:c({M,F,A}).
```

The function `hipe:c/2` can also be used, which takes the list of the HiPE compiler options as its second argument.

Finally, should you forget everything else, you can always type the following from the ERLANG shell:

```
2> hipe:help().
```

which will display a short user's guide to the HiPE compiler.

2.3 HiPE compiler options

For the average user, it should not be necessary to give any extra information to the compiler than described in the previous section. However, in some cases it may be useful or even necessary to control the behavior of the native code compilation. To pass options to the HiPE compiler via the normal ERLANG compiler interface, these must be wrapped in a term {`hipe, ...`}. For example:

```
3> c(my_module, [native, {hipe, [verbose, o3]}]).
```

will pass the flags `verbose` and `o3` to the HiPE compiler. Note that if only a single option is given, it does not have to be wrapped in a list, as in e.g.:

```
c(my_module, [native, {hipe, verbose}]).
```

The main useful options are the following:

`o0, o1, o2, o3` Selects the optimization level, `o0` being the lowest. The default is `o2`. Upper case versions of these options also exist, i.e., `O2` is an alias for `o2`, etc.

`verbose` Enables HiPE compiler verbosity. Useful if you want to see what is going on, identify functions whose native code compilation is possibly a bottleneck, or just check that the native code compiler is running.

If a module takes too long time to compile, try using a lower optimization level such as `o1`. You can also try keeping the current optimization level, but specifically select the faster but less precise *linear scan* algorithm for register allocation [7]. (Register allocation is one of the major bottlenecks in the optimizing native code compilers.) This is done by adding the option `{regalloc,linear_scan}`, as in:

```
c(my_module, [{hipe, [{regalloc,linear_scan}]}]).
```

If you wish to always use certain HiPE compiler options for some particular module, you can place them in a `compile` directive in the source file, as in the following line:

```
-compile({hipe, [o1]}).
```

Note: options early in the list (i.e., further to the left) take precedence over later options. Thus, if you specify e.g.

```
{hipe, [o3, {regalloc,linear_scan}]}
```

the `o3` option will override the `regalloc` option with the more advanced (and more demanding compilation-time wise) `o3`-level iterated coalescing register allocator. The correct way would be:

```
{hipe, [{regalloc,linear_scan}, o3]}
```

which specifies `o3`-level optimizations but with fast register allocator.

More information on the options that the HiPE compiler accepts can be obtained by:

```
hipe:help_options().
```

2.4 Recent improvements

2.4.1 Local type propagator

ERLANG, being a dynamically typed language, often provides the developer with freedom to experiment with data structures whose handling is possibly still incomplete, and rapidly prototype applications. However, this also means that a lot of run time is spent in performing type tests (that usually succeed) to ensure that the operations performed are meaningful, e.g., that a program does not accidentally succeed in dividing a float by a list or taking the fifth element of a process identifier.

One of the recent additions to the HiPE compiler is a *local type propagator* which tries to discover as much of the available (per-function) type information as possible at compile time. This information is then propagated throughout the code of the function to eliminate redundant type tests and to transform polymorphic primitive operations that operate on general types into faster operations that are specialized to the type of operands actually being used.

Since the type propagator is a recent addition that is still under development and further extensions of its functionality are underway, we have not yet conducted a proper evaluation of the time performance improvements that one can expect from it in practice. However, preliminary numbers indicate that the size of the native code is noticeably reduced, something which in turn has positive effects on the later optimization passes, often resulting in compile times even shorter than those of the HiPE compiler in R9B.

The type propagator is enabled by default at the normal optimization level `o2` (or higher).

2.4.2 Handling of floats

In the runtime system, atomic ERLANG values are represented as tagged 32-bit words; see [10]. Whenever a tagged value is too big to fit into one machine word the value is *boxed*, i.e., put on the heap with a header word preceeding it which is pointed to by the tagged value. Floating point numbers have 64-bit precision and are therefore typically boxed. This means that whenever they need to be used as operands to a floating point operation, they need to be unboxed, and after the operation is performed the result must then be boxed and stored back on the heap.

To avoid this overhead, starting from R9B, the BEAM has been enhanced with special floating point instructions that operate directly on untagged values. This has sped up the handling of floats considerably since the number of boxing/unboxing operations are reduced. However, since the BEAM code is interpreted, floating point arithmetic is still not taking advantage of features available at the floating point unit (FPU) of the target architecture, such as machine registers. More specifically, the operands are put into the FPU, the operation is performed, and then the result is taken out and stored in memory.

In the HiPE compiler, floating point values are mapped to the FPU and are kept there for as long as possible, eliminating even more overhead from floating point calculations. In [8] we have described in detail the two back-end specific schemes used in the mapping. Our performance comparison shows that HiPE-compiled floating point intensive code can be considerably faster than floating-point aware BEAM bytecode. Table 1 gives an idea of the performance improvements that can be expected across a range of programs manipulating floats.

To maximize the gain of the floating point instruction the user is encouraged to use appropriate `is_float/1` guards that currently communicate to the BEAM compiler the floating point type information[2] and to try to keep floating point arithmetic instructions together in blocks, i.e., not split them up by inserting other instructions that can just as well be performed before or after the calculations.

The more efficient, target-specific compilation of floating point arithmetic is enabled by default starting at optimization level `o1`.

2.4.3 Handling of binaries

Proper support for the bit syntax [9] was introduced into Erlang/OTP in R8B. Initially, the HiPE compiler used a rather naïve compilation scheme: binary matching instructions of the BEAM were translated into calls to C functions which were part of the interpreter's supporting routines. As a result, the HiPE-compiled code was actually slightly slower than the BEAM code because of costs in switching between native and interpreted code (cf. also Section 3). To remedy this, we proposed and implemented a scheme that relies on a *partial translation* of binary matching operations. This scheme identifies special "common" cases of binary match-

[2]Explicitly writing such guards will become unnecessary when the global type analysis gets fully integrated in HiPE; see Section 2.5.2.

Table 1: Performance of BEAM and HiPE in R9B on programs manipulating floats (times in ms).

Benchmark	BEAM	HiPE	speedup
float_bm	14800	4040	3.66
barnes-hut	10250	4280	2.39
fft	16740	8890	1.88
wings	8310	7370	1.12
raytracer	9110	8500	1.07
pseudoknot	3110	1440	2.16

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
float_bm	1930	750	2.57
barnes-hut	1510	600	2.51
fft	2830	1450	1.95
wings	1160	850	1.36
raytracer	1200	1070	1.12
pseudoknot	380	140	2.71

(b) Performance on x86.

ings and translates these completely into native code, while the remaining "uncommon" cases still call C functions in order to avoid extensive code bloat. The implementation of this compilation scheme is described in [4] and is included in the HiPE compiler as of the R9B release of Erlang/OTP.

The performance of this scheme on several different benchmarks involving binaries is shown in Table 2. The first three benchmarks test the speed of binary matching: the **bsextract** benchmark takes a binary containing a GTP_C message as input, extracts the information from the message header, and returns it. The **bsdecode** benchmark is similar but rather than simply extracting a binary, it also translates the entire message into a record. The **ber_decode** benchmark, generated by the ASN.1 compiler, parses a binary. The last two benchmarks, **bsencode** and **ber_encode**, test the speed of binary creation rather than matching.

As expected, speedups are obtained when there is information available at compile time to identify cases which can be compiled fully to native code. Such, for example, is the case when the binary segment sizes are constant or when it is possible to determine statically that a segment of a binary starts at a byte boundary. In other words, to achieve best time performance, it might be advisable to use some extra space to guarantee that each element starts at a byte boundary. For example, if one wants to use binaries to denote 16 integers where each integer needs 7 bits it is possible to pack them so that they only take up 14 bytes. If each integer is put a byte boundary, the binary will take up more space (16 bytes), but the binary matching operations will be performed faster.

The HiPE compiler option `inline_bs` enables native code compilation of the bit syntax. This option is selected by default at optimization level `o1` or higher, and the only reasons for a user to disable it is either to test its performance effects or if code size is a serious concern.

2.5 Planned extensions for the near future

2.5.1 Better translation of binaries

The compilation scheme introduced in R9B made binary matching faster, but more work has since been done to make it even faster. In the upcoming R9C release, a new scheme for compiling binary matching will be included. Rather than relying on a partial translation and having BEAM be in control, the entire binary matching operation will fall in the hands of the native code compiler. This has made it possible to avoid several unnecessary mode switches. With this scheme most binary matching code will make no calls to the C functions which are used strictly as a last resort when a single operation becomes very complex.

In addition to this a new scheme to compile binary creation has been developed. It is developed in a similar fashion to the binary matching scheme by changing calls to C functions into specialized versions of these functions that are then translated into native code.

As seen in Table 3, the performance of HiPE compiled code has been improved substantially. The speedup for the matching benchmarks ranges from 1.5-4 times compared to BEAM. The speedup for benchmarks that create binaries is more than 2 times on x86 and more than 1.5 times on SPARC.

In connection with the effort to compile directly from Core Erlang to native code a project has started to further improve the compilation of binary matching as new possibilities open up when the structure of the matching becomes visible to the compiler. The result of this project will likely be available in the R10 release.

2.5.2 Global type analysis

As described in Section 2.4.1, the HiPE compiler now includes a local type propagator which works on individual functions. However, if no assumptions can be made about the arguments to the functions, only the most basic type information can be found. We have therefore implemented a *global type analyzer* which processes a whole module at a time. It can generally find much more detailed type information, but the precision depends to a large extent on the programming style of the analyzed code. Since an exported function can potentially be called from anywhere outside the module and with any inputs, it is not possible to make any assumptions about the types of the arguments to exported functions. The best precision is achieved when only the necessary interface functions are exported, and the code does all or most of its work within the same module. When module boundaries are crossed, type information is lost. For most built-in functions, however, we can know what types of data they accept as input and what types they return.

We are currently working on how to take advantage of the gathered type information (in combination with the local type propagator). First of all, we are often able to remove unnecessary type checks from the code. Second, it is sometimes possible to avoid repeatedly tagging and untagging values (cf. Section 2.4.2). Third, global type analysis makes it possible to avoid creating tuples for returning multiple values from a function, when the result of the function is always immediately unpacked – instead, multiple values can be passed directly in registers or on the stack.

Note that the global type analysis is *not a type checker*

Table 2: Performance of BEAM and HiPE in R9B on programs manipulating binaries (times in ms).

Benchmark	BEAM	HiPE	speedup
bsextract	15540	8450	1.84
bsdecode	27070	26860	1.01
ber_decode	14350	9130	1.57
bsencode	14210	15870	0.90
ber_encode	18720	16280	1.15

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
bsextract	14500	7350	1.97
bsdecode	13490	12970	1.04
ber_decode	16500	9200	1.79
bsencode	17540	16030	1.09
ber_encode	21870	17420	1.26

(b) Performance on x86.

Table 3: Performance of BEAM and HiPE in a pre-release of R9C on programs manipulating binaries.

Benchmark	BEAM	HiPE	speedup
bsextract	13380	4060	3.30
bsdecode	26060	21110	1.23
ber_decode	13980	5720	2.44
bsencode	16960	11070	1.53
ber_encode	18150	9510	1.91

(a) Performance on SPARC.

Benchmark	BEAM	HiPE	speedup
bsextract	14700	3560	4.13
bsdecode	13670	7780	1.76
ber_decode	15610	6070	2.57
bsencode	16790	7290	2.30
ber_encode	22560	10860	2.08

(b) Performance on x86.

or type inference system, i.e., the user is not able to specify types (because the user cannot be completely trusted), and furthermore, the fact that an input parameter is always *used* as e.g. an integer does not mean that the passed value will always *be* an integer at runtime. Indeed, the current implementation does not even give a warning to the user if it detects a type error in the program, but just generates code to produce a runtime type error. This might change in the future, to make the type analyzer useful also as a programming tool.

2.5.3 Compilation from Core Erlang

A new feature of the HiPE compiler is the ability to compile to native code directly from the ERLANG source code, (i.e., instead of starting with the BEAM virtual machine code, which was previously the only way). This is done by generating HiPE's intermediate ICode representation directly from the Core Erlang code which is produced by the Erlang/OTP compiler. No BEAM code needs to have been previously generated. The advantages of this are better control over the generated code, and greater ability to make use of metadata about the program gathered on the source level, such as global type analysis information.

Currently, the way to do this is to add an extra option `core` when compiling to native code:

```
c(my_module, [native, core]).
```

However, this method of compiling is not yet fully functional in the coming R9C release, in that some programming constructs are not yet handled properly. We intend to have compilation from source code completely implemented in release R10.

We expect that in the future, compilation from source code will be the default method of the HiPE compiler. The compilation from BEAM will however still be available, for those cases when the source code is not available, or it is for other reasons not possible to recompile from the sources.

3. RECOMMENDATIONS ON HIPE'S USE

3.1 Improving performance from native code

- If your application spends most of its time in known parts of *your* code, and the size of those parts is not too large, then compiling those parts to native code will maximize performance.
- Largish self-contained modules with narrow external interfaces allow the compiler to perform useful module-global type analysis and function inlining, when compiling via Core Erlang.
- While very deep recursions are not recommended, they are much more efficient in native code than in BEAM code. This is because the HiPE runtime system includes specific optimizations (*generational stack scanning* [3, 11]) for this case.
- Monomorphic functions, functions that are known to operate on a single type of data, are more likely to be translated to good code than polymorphic functions. This can be achieved by having guards in the function heads, or by avoiding to export the functions and always calling them with parameters of a single type, known through guards or other type tests.
- When using floating point arithmetic, collect the arithmetic operations in a block and try to avoid breaking it up with other operations; in particular try to avoid calling other functions. Help the analysis by using the guard `is_float/1`. You might still benefit from this even if you do not manage to keep the operations in a block; the risk of losing performance is minimal.
- Order function and case clauses so that the cases that are more frequent at runtime precede those that are less frequent. This can help reduce the number of type tests at runtime.

3.2 Avoiding performance losses in native code

- If the most frequently executed code in your application is too large, then compiling to native code may give only a small or even negative speedup. This is because native code is larger than BEAM code, and in this case may suffer from excessive *cache misses* due to the small caches most processors have.
- Avoid calling BEAM-code functions from native-code functions. Doing so causes at least two mode switches (one at the call, and one at the return point), and these are relatively expensive. You should native-compile *all* code in the most frequently executed parts, including Erlang libraries you call, otherwise excessive mode switching may cancel the performance improvements in the native-compiled parts.
- Do *not* use the `-compile(export_all)` directive. This reduces the likelihood of functions being inlined, and makes useful type analysis impossible.
- Avoid crossing module boundaries too often (making remote calls), since the compiler cannot make any assumptions about the functions being called. Creative use of the pre-processor's `-include` and `-define` directives may allow you to combine performance and modular code.
- Avoid using 32-bit floats when using the bit syntax, since they always require a mode switch. It is also costly to match out a binary that does not start at a byte boundary, mainly because this requires that all the data of the binary is copied to a new location. If on the other hand a binary starting at a byte boundary is matched, a sub-binary which only contains a pointer to the data is created. When variable segment lengths are used it is beneficial to have a unit that is divisible by 8, because this means that byte boundary alignment information can be propagated.

3.3 Cases when native code does not help

- Be aware that almost all BIF calls end up as calls to C functions, even in native code. If your application spends most of its time in BIFs, for instance accessing ETS tables, then native-compiling your code will have little impact on overall performance.
- Similarly, code that simply sends and receives messages without performing significant amounts of computation does not benefit from compilation to native code; again, this is because the time is mostly spent in the runtime system.

4. THE TRUTH, THE WHOLE TRUTH, AND NOTHING BUT THE TRUTH

Significant work has been put into making HiPE a robust, "commercial-quality" compiler.[3] As a matter of fact, we have mostly tried to follow BEAM's decisions in order to preserve the observable behavior of ERLANG programs, even if that occasionally meant possibly reduced speedups in performance. Still, a couple of small differences with code produced by BEAM exist, and the user should be aware of some limitations. We document them below.

[3]At the time of this writing, July 2003, we are not aware of any outstanding bugs.

4.1 Incompatibilities with the BEAM compiler

- Detailed stack backtraces are currently not generated from exceptions in native code; however, where possible, the stack trace contains at least the function where the error occurred. Performing pattern matching on stack backtraces is not recommended in general, regardless of the compiler being used.
- The old-fashioned syntax `Fun = {M,F}, Fun(...)` for higher-order calls is not supported in HiPE. In our opinion, it causes too many complications, including code bloat. Proper `fun`s should be used instead, or explicit calls `M:F(...)`.
- On the x86, floating-point computations may give different (more precise) results in native code than in BEAM. This is because the x86 floating-point unit internally uses higher precision than the 64-bit IEEE format used for boxed floats, and HiPE often keeps floats in the floating-point unit when BEAM would store them in memory; see [8].

4.2 Current limitations

- Once inserted into the runtime system, native code is never freed. Even if a newer version of the code is loaded, the old code is also kept around.
- The HiPE compiler recognizes literals (constant terms) and places them in a special *literals area*. Due to architectural limitations of the current Erlang/OTP runtime system, this is a single area of a fixed size determined when the runtime system is compiled. Loading a lot of native code that has many constant terms will eventually cause the literals area to fill up, at which point the runtime system is terminated. A short-term fix is to edit `hipe_bif0.c` and explicitly make the literals area larger.

5. CONCLUDING REMARKS

We have presented a user-oriented description of features and characteristics of the HiPE native code compiler, which nowadays is integrated in Erlang/OTP and easily usable by ERLANG application developers and aficionados. We hold that HiPE has a lot to offer to its users. Some of its benefits are described in this paper. Others, perhaps more exciting ones, await their discovery.

One final word of advice: HiPE, like any compiler, can of course be treated as a "black-box", but we generally recommend to creatively explore its options and flexibility and add color to your life!

6. ACKNOWLEDGMENTS

The HiPE compiler would not have been possible without the prior involvement of Erik "Happi" Stenman (formerly Johansson) in the project. Its integration in Erlang/OTP would still be a dream without close collaboration with members of the Erlang/OTP group at Ericsson (Björn Gustavsson, Kenneth Lundin, and Patrik Nyblom), and the active encouragement of Bjarne Däcker. HiPE's development has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Utvecklings AB.

7. REFERENCES

[1] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[2] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, Nov. 2000.

[3] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 162–173. ACM Press, 1998.

[4] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 6–15. ACM Press, Nov. 2002.

[5] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.

[6] 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*, 2003. To appear.

[7] E. Johansson and K. Sagonas. Linear scan register allocation in a high performance Erlang compiler. In *Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium*, number 2257 in LNCS, pages 299–317. Springer, Jan. 2002.

[8] T. Lindahl and K. Sagonas. Unboxed compilation of floating point arithmetic in a dynamically typed language environment. In R. Peña and T. Arts, editors, *Implementation of Functional Languages: Proceedings of the 14th International Workshop*, number 2670 in LNCS, pages 134–149. Springer, Sept. 2002.

[9] P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.

[10] M. Pettersson. A staged tag scheme for Erlang. Technical Report 029, Information Technology Department, Uppsala University, Nov. 2000.

[11] 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*, number 2441 in LNCS, pages 228–244. Springer, Sept. 2002.

[12] J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, June 1991.

[13] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In A. Mycroft, editor, *Proceedings of the Second Static Analysis Symposium*, number 983 in LNCS, pages 366–381. Springer, Sept. 1995.

[14] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.