National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

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

Diploma Thesis

KONSTANTINOS KALLAS

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Athens, July 2018

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

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

Diploma Thesis

KONSTANTINOS KALLAS

Supervisor : Konstantinos Sagonas
Associate Professor NTUA

Approved by the examining committee on the July 6, 2018.

....	....	....
Konstantinos Sagonas	Nikolaos S. Papaspyrou	Nectarios Koziris
Associate Professor NTUA	Associate Professor NTUA	Professor NTUA

Athens, July 2018

....

Konstantinos Kallas
Electrical and Computer Engineer

Abstract

We introduce HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on HiPE, the High Performance Erlang compiler. HiPErJiT uses runtime profiling to decide which modules to compile to native code and which of their functions to inline and type-specialize. HiPErJiT is integrated with the runtime system of Erlang/OTP and preserves aspects of Erlang's compilation which are crucial for its applications: most notably, tail-call optimization and hot code loading at the module level. We present HiPErJiT's architecture, describe the optimizations that it performs, and compare its performance with BEAM, HiPE, and Pyrlang. HiPErJiT offers performance which is about two times faster than BEAM and almost as fast as HiPE, despite the profiling and compilation overhead that it has to pay compared to an ahead-of-time native code compiler. But there also exist programs for which HiPErJiT's profile-driven optimizations allow it to surpass HiPE's performance.

Key words

JiT compiler, HiPE, Erlang, profiling, type specialization, profile-driven optimizations

Acknowledgements

First of all, I would like to thank my advisor Kostis Sagonas, for his precious guidance throughout my thesis, as well as for his willingness to discuss my every question. In addition, I would like to express my gratitude towards Nikos Papaspyrou, because the courses that he taught together with Kostis were my very first and most crucial exposure to the field of programming languages.

I would also like to thank my peers and friends for all our intriguing discussions, which made me reflect, shaped me, and contributed to the most crucial decisions of my life. On top of that, they colored my life with the most amazing experiences.

Finally, I would like to thank my parents, Yiannis and Katerina, and my brother, Nikos, who always stood by my side, put all their trust in me, and supported all my choices as if they were theirs.

Konstantinos Kallas,

Athens, July 6, 2018

This thesis is also available as Technical Report CSD-SW-TR-4-18, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2018.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Erlang is a concurrent functional programming language with features that support the development of scalable concurrent and distributed applications, and systems with requirements for high availability and responsiveness. Its main implementation, the Erlang/OTP system, comes with a byte code compiler for its virtual machine, called BEAM, which produces portable and reasonably efficient code. For applications with requirements for better performance, an ahead-of-time native code compiler called HiPE (High Performance Erlang) can be selected. In fact, byte code and native code can happily coexist in the Erlang/OTP runtime system.

Despite this flexibility, the selection of the modules of an application to compile to native code is currently manual. Perhaps it would be better if the system itself could decide on which parts to compile to native code in a just-in-time fashion. Moreover, it would be best if this process was guided by profiling information gathered during runtime, and was intelligent enough to allow for the continuous run-time optimization of the code of an application.

1.1 Thesis goals

In this thesis we will describe the design and implementation of HiPErJiT, a profile-driven Just-in-Time compiler for Erlang based on HiPE. We believe that it is a solid first step towards a continuous run-time optimizing compiler. Our main goals are that HiPErJiT:

- Achieves performance benefits over other Erlang compilers.
- Preserves all the important features of Erlang.
- Is easy to maintain and keep in-sync with other Erlang/OTP components.

HiPErJiT employs the tracing support of the Erlang runtime system to profile the code of bytecode-compiled modules during execution and choose whether to compile these modules to native code, currently employing all optimizations that HiPE also performs by default. For the chosen modules, it additionally decides which of their functions to inline and/or specialize based on runtime type information. We envision that the list of additional optimizations to perform based on profiling information will be extended in the future, especially if HiPErJiT grows to become a JiT compiler which performs lifelong feedback-directed optimization of programs. Currently, JiT compilation is triggered only once for each loaded instance of a module, and the profiling of their functions is stopped at that point.

1.2 Thesis outline

The thesis is organized into the following chapters.

- Chapter 2 provides background information related to Erlang, the Erlang Run-Time System, the various Erlang compilers, and previous work on Just-in-Time compilation.
- Chapter 3 presents the architecture of HiPErJiT and the rationale behind some of our design decisions.
- Chapter 4 describes the profile-driven optimizations that HiPErJiT performs.
- Chapter 5 presents a performance evaluation of HiPErJiT compared to other Erlang compilers.
- Chapter 6 describes the current state of HiPErJiT and proposes possible future work.

Chapter 2

Background

2.1 Erlang, ERTS, and HiPE

Before describing our work, it is crucial that the reader understand the fundamentals of Erlang, its runtime system, and its standard ahead of time compiler, namely HiPE.

2.1.1 Erlang

Erlang is a dynamically typed, strict, concurrency oriented, functional language. It is designed to be used for applications in telecommunication systems. Because of that, it was designed to satisfy six essential requirements [Arms03].

- Concurrency: Very small overhead to create, destroy, and maintain a large number of concurrent processes.
- Error Encapsulation: Errors occurring in one process should never damage other system processes.
- Fault Detection: It must be possible to detect errors both locally (in the same process) or remotely (in a different process).
- Fault Identification: It must be possible to identify the cause of an error, so that corrective action can be taken.
- Code Upgrade: It should be possible to change code as it is executing, without stopping or restarting the system.
- Stable Storage: It should be possible to store data in a manner that is resilient against system crashes.

The Erlang view of the world is that everything is a *process* and that each *process* is isolated, shares no resources, and interacts with each environment only[1] through message passing.

As there are no shared data structures, monitors, locks, etc... in Erlang it is easy to understand how concurrency works after understanding Erlang's sequential subset. This subset is a dynamically typed, strict, functional programming language, which is largely free from side effects.

Sequential Erlang

Erlang has eight primitive data types.

[1] Technically there also exist other ways for processes to communicate, such as the Erlang Term Storage (ETS) which allow for data to be accessed from all processes in a system.

- Integers of unlimited precision. Examples: `42, 2#101, 16#1f`.
- Atoms are used to denote distinguished values. They are sequences of alphanumeric characters beginning with a small letter. In case they contain spaces or capital letters they must be enclosed in single quotes. Examples: `hello, phone_number, 'Monday'`.
- Floats are represented as IEEE 754 64 bit floating point numbers [IEEE08]. Examples: `2.3, 2.3e3, 2.3e-3`.
- References are globally unique symbols whose only property are that they can be compared for equality. They are created by calling the function `make_ref/0`.
- Binaries are sequences of bytes. They are used to efficiently store binary data. Examples: `<<10,20>>, <<"ABC">>, <<1:1,0:1>>`.
- Pids are process identifiers, that are used to reference processes. Example: `<0.51.0>`.
- Port Identifiers are similar to Pids, but are used to identify ports, which are the basic mechanism for the communication with the external world. They are created by calling the function `open_port/2`.
- Funs are function closures, also known as lambda expressions in other languages. Example: `fun (X) -> X+1 end`.

Erlang also supports three compound data types.

- Tuples are fixed size containers of Erlang data types. Example: `{adam,24,{july,29}}`.
- Lists are variable size containers of Erlang data types. They are generally similar to list data types in other functional programming languages. A very useful operation on lists is cons `|` which separates its head and its tail. Examples: `[], [a,2,{c,4}], [1|[2|[]]]`.
- Maps contain a variable number of key-value associations, where the key and the value are Erlang data types. Each association is called an association pair, and the number of association pairs is called the size of the map. Example: `#{name=>adam,age=>24,date=>{july,29}}`.

There are also two main forms of syntactic sugar regarding those data types.

- Strings are simple lists of integer ASCII codes. Examples: `"hello", [104,101,108,108,111]`.
- Records allow for the elements of a tuple to be assigned a name so that they can be referenced by name and not by position. Example: `#person{name=adam,age=24,date={july,29}}`.

Erlang contains variables as most other languages. They can be thought as single assignment variables in other standard languages. Practically they are bound to a value and then represent it for the rest of the program. They are written using a sequence of characters starting with an uppercase letter or the character underscore _. Examples: `Var, _V, _`.

Variables in Erlang are bound to values using a Pattern Matching mechanism, which is based on Erlang terms and patterns. In Erlang a term is defined recursively as either a primitive data type or a tuple of terms or a list of terms or a map of terms. Practically a term in Erlang is any data type.

A Pattern is similarly defined recursively as either a primitive data type or a variable or a tuple of patterns or a list of patterns or a map of patterns. A primitive pattern is a pattern where each variable occurs just once.

After defining terms and patterns, Pattern Matching can be thought as a recursive comparison between a pattern and a term. Three requirements must be met for a pattern match to succeed.

- The pattern must be primitive.
- The pattern must be either equal to the term, or contain a variable in the position where the term contains an inner term.
- Each variable should be either bound to a term equal to its corresponding term or should be unbound. In case it is unbound, the pattern match leads to a consequent bounding between the variable and the corresponding term.

Below are some examples of successful and unsuccessful pattern matches.

```
%% Successful pattern matches
1 = 1.
X = 2.
{Y, Z} = {1,3}.
[A, 2] = [42, 2].

%% Unsuccessful pattern matches
{B, 1} = {1,2}.
[] = [1].
```

Listing 2.1: An example of successful and unsuccessful pattern matches.

It is sometimes useful to extend a pattern match with a set of constraints on the matched pattern. This is achieved by using guards, which are expressions that refine patterns with predicates. They are introduced with the **`when`** keyword and they can only contain a sequence of binary operators, such as `<`, `>`, ..., and some built-in functions. Below are some examples of patterns extended with guards.

```
{Tag, Message} when Tag /= quit
Number when Number >= 0
```

Listing 2.2: An example of guarded patterns.

The most fundamental Erlang construct is functions. Functions in Erlang are defined similarly to how they are defined in other functional languages. A function definition contains clauses, each of whom is composed of a head and a body. The head contains patterns and guards and the body contains a sequence of expressions.

```
FunctionName(P11, ..., P1N) when G11, ..., G1M ->
  Body1;
FunctionName(P21, ..., P2N) when G21, ..., G2M ->
  Body2;
...
FunctionName(PK1, ..., PKN) when GK1, ..., GKM ->
  BodyK.
```

Listing 2.3: An example of a function definition.

In the above example `FunctionName` is an atom, `P11, ..., PKN` are patterns, `G11, ..., GKM` are guards, and `Body1, ..., BodyK` are sequences of expressions.

In order to evaluate a function call `Fun(Arg1, Arg2, ..., ArgN)` a satisfying definition has to be found. The satisfying definition is the first one where the function name is the same `Fun = FunctionName` and all arguments match to the guarded patterns `Arg1 = Pi1` **`when`** `Gi1, ...`.

After finding the satisfying definition, all free variables occuring in patterns of it are bound with the actual arguments based on the rules that were discussed above. Then the expressions of the body are evaluated. The function call then returns the value of the last expression in its body.

```
member(H, [H1|_]) when H == H1 -> true;
member(H, [_|T])  -> member(H, T);
member(H, [])     -> false.
```

Listing 2.4: An example of a member function definition

The function call `member(dog, [cat,man,dog,ape])` would be evaluated as follows:

- The first clause matches with bindings $\{$`H` $\mapsto$ `dog`, `H1` $\mapsto$ `cat`$\}$. The guard test then fails.
- The second clause matches with $\{$`H` $\mapsto$ `dog`, `T` $\mapsto$ `[man,dog,ape]`$\}$. There are no guards so the `member(H, T)` is evaluated.
- Evaluate `member(dog, [man,dog,ape])`
- As before the second clause matches with $\{$`H` $\mapsto$ `dog`, `T` $\mapsto$ `[dog,ape]`$\}$.
- Evaluate `member(dog, [dog,ape])`
- The first clause matches with $\{$`H` $\mapsto$ `dog`, `H1` $\mapsto$ `dog`$\}$ and the guard test succeeds.
- Evaluate **`true`** and return **`true`**.

An important characteristic of functions in Erlang is that they can be tail-recursive. A tail-recursive function is a function whose final expressions in its bodies are either variables or function calls, also known as tail-calls. Let us clarify that with an example:

```
%% Not tail recursive as one body ends with N * factorial(N-1)
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

%% A tail recursive way of writing factorial
factorial(N) -> factorial_1(N, 1).
factorial_1(0, X) -> X;
factorial_1(N, X) -> factorial_1(N-1, N*X).
```

Listing 2.5: An example of a tail-recursive factorial definition

The importance of tail recursion is that it can sufficiently replace loop constructs that are present in other languages. As Erlang does not contain any loops and while constructs, tail-recursion is the only way to achieve this behaviour. In order to understand this, consider the following simple example:

```
p() ->
  ...
  q(),
  ...

q() ->
  r(),
  s().
```

Listing 2.6: An example of tail recursion used as a simple loop

At some point during the evaluation of the body p, function q is called. The final expression in the body of q is a function call to s. s will return a value to q, and q will just return it unmodified to p. Function calls are normally compiled to code that also keeps a return address somewhere (usually in the stack), thus memory limits disallow infinite function calls as the stack will eventually become full. As q does not modify the returned value in some way, there is no need to keep the return address q and a tail-call can be compiled into a simple jump instruction [Stee77]. Because of that, tail-recursive functions can replace loops without consuming unnecessary memory.

Erlang also contains a set of Built-In Functions (BIFs), which are implemented in C code in the runtime system. They implement useful behaviour that is usually too low level to implement in Erlang. For example:

```erlang
atom_to_list(erlang).
is_float(Var).
```

Listing 2.7: Example BIFs

Erlang code is divided into modules. Modules consist of a sequence of attributes and function definitions.

```erlang
-module(factorial).  % module attribute
-export([fact/1]).   % module attribute

fact(N) when N>0 ->  % beginning of function declaration
  N * fact(N-1);     %  |
fact(0) ->           %  |
  1.                 % end of function declaration
```

Listing 2.8: A module containing a factorial function

Module attributes define certain properties of the module. They consist of a tag and a value. In the above example there are two attributes. The first one, with tag **module** and value factorial, indicates the name of the module. The second one indicates which functions are exported from the module, which means which functions can be called by code outside of this module. In order to call a function from an external module, the function call must also contain the module name. Example: factorial:fact(5) instead of just fact(5).

An important note is, that there also exist other predefined module attributes, such as macro and type definitions. Users can also define their own module attributes.

Concurrent Erlang

After briefly describing the sequential subset of Erlang, it is possible to move on to its concurrent part. As also noted above, Erlang is based on isolated processes, which interact through message passing.

An Erlang process is a self-contained separate unit of computation, which exists concurrently with other processes in the same system. There is no inherent hierarchy among processes, however the developer can explicitly impose one if they wish.

Creating processes is done by calling the BIF **spawn**/3, which creates a new process and starts its execution, using the given function and arguments. A call to **spawn** returns immediately after the new process is created and does not wait until the evaluation of the starting function. When the evaluation of the starting function finishes, the spawned process is terminated automatically.

In Erlang processes only communicate between each other via message passing. Messages are sent to a process with Pid using the primitive send operation Pid ! Message.

Send (`!`) first evaluates its arguments and returns the message sent after evaluation. The send command is asynchronous, so it will return immediately and will not wait for the message to arrive to the destination or be received. The system will not notify the sender if the process to which the message is being sent, has already terminated. The application must itself implement all forms of checking. The system only ensures that messages are always delivered to the recipient, and that they are always delivered in the same order that they were sent from the same sender to the same receiver. For example if two messages `Msg1, Msg2` are sent from process A to process B in that order, they will always arrive to B in that order.

Receiving messages is achieved using the primitive **`receive`**, which has the following syntax:

```
receive
  Message1 [when Guard1] ->
    Actions1;
  Message2 [when Guard2] ->
    Actions2;
  ...
end
```

Listing 2.9: The syntax of receive

Every process has a mailbox where all messages that are sent to the process are stored in the order that they arrived. A **`receive`** checks all patterns `Message1, Message2, ...` and tries to match them against messages in the process' mailbox. When a matching message `MessageN` is found and the corresponding guard is satisfied, the message is removed from the mailbox and then `ActionsN` are evaluated. Pattern matching works as described above. Any messages left in the mailbox, that are not selected, will remain in the mailbox in the same order that they were delivered and they will be matched against the next **`receive`**. The process evaluating **`receive`** will block until a matching message is found in the mailbox.

It is possible to implement a selective receive mechanism in Erlang, however as messages not matched are left in the mailbox, it is the application's responsibility to make sure that the system does not fill up with unmatched messages.

Here is an example showing the message receiving order in Erlang. It is taken from [Vird96].

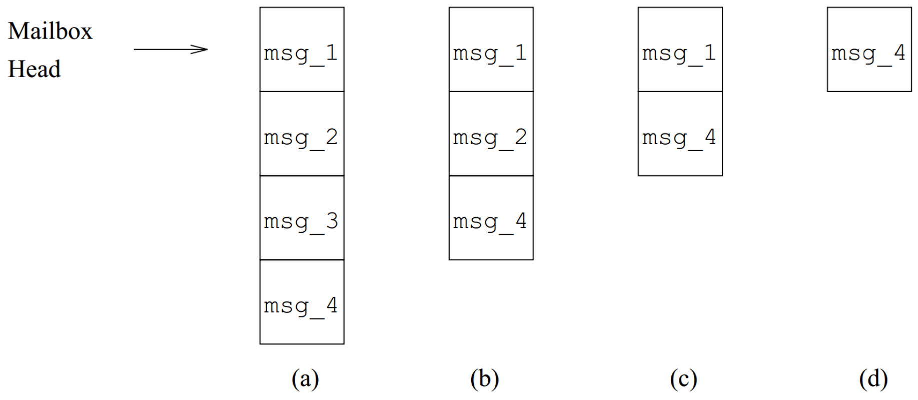

Figure 2.1: Message Reception

A process mailbox initially contains four messages (Fig. 2.1 (a)) `msg_1, msg_2, msg_3, msg_4` in this order.

Then the following **`receive`** is evaluated:

```erlang
receive
  msg_3 ->
    ...
end
```

Listing 2.10: First evaluated receive

The result is that `msg_3` is matched and subsequently removed from the mailbox, as shown in Fig. 2.1 (b).

When the following receive is evaluated:

```erlang
receive
  msg_4 ->
    ...
  msg_2 ->
    ...
end
```

Listing 2.11: Second evaluated receive

This will try to match each message in the mailbox against `msg_4` and then against `msg_2`. This results in `msg_2` being matched and removed, as shown in Fig. 2.1 (c).

Finally evaluating:

```erlang
receive
  AnyMessage ->
    ...
end
```

Listing 2.12: Third evaluated receive

where `AnyMessage` is unbound, results in `msg_1` being matched and removed from the mailbox, as shown in Fig. 2.1 (d).

2.1.2 Erlang Run Time System - ERTS

The Erlang Runtime System is complex and contains many independent components. There is no official definition of what an Erlang Runtime System is, so we will focus on the *de facto* Erlang Runtime System implementation "Erlang/OTP", which is developed and maintained by Ericsson. We will call it ERTS (Erlang Run Time System).

An Erlang application or system is usually just a node that runs ERTS and the BEAM virtual machine. According to the Erlang/OTP documentation, a node is an executing runtime system that has a given name.

The layers that an Erlang application runs on are shown in Fig. 2.2 [2].

We need to make sure that ERTS satisfies the requirements that we discussed in Section 2.1.1. As we pointed out Erlang is a concurrency oriented language and so it must maintain a large number of concurrently executing processes. To achieve that, those processes must be scheduled in an efficient way. More specifically, the two following criteria should be satisfied [Vird96]:

- The scheduling algorithm should be fair, so that every process which can be run will be run. Preferably the processes will run in the order in which they became runnable.

[2] Taken from https://github.com/happi/theBeamBook

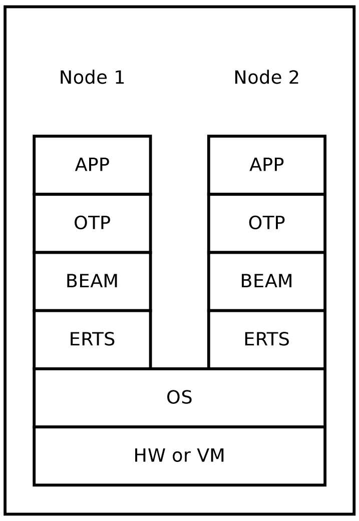

Figure 2.2: ERTS Stack

- No process will be allowed to block for a long time. Every process is allowed to run for a short period of time, in Erlang known as a time slice, before it is rescheduled to allow another runnable process to be run.

Time slices are typically set to allow the executing process to perform a fixed number of reductions[3], usually around 2000, before being rescheduled.

Erlang should be suitable for soft real-time applications and because of that, response times should be in the order of milliseconds. A scheduling algorithm that meets the above criteria is considered good enough for an Erlang implementation [Vird96].

As Erlang hides all memory management from the user, it is important that automatic memory management does not violate the criteria mentioned above. In essence automatic memory management should be done in a manner as not to block the system for a big length of time, preferably for a shorter time than the time slice of a single process.

Before diving in ERTS and its functionality we will give a brief overview of its components and define some necessary vocabulary.

An Erlang process is similar to an OS process. Each has its own memory (stack, heap, mailbox) and a Process Control Block (PCB) with process information. All Erlang code execution is done within the context of a single process. Processes only communicate with each other through message passing. This is even true for processes in different Erlang nodes.

The Erlang compiler, as its name implies, compiles Erlang source code, contained in `.erl` files, into virtual machine bytecode for BEAM (the virtual machine). The compiler is itself written in Erlang and compiled by itself to BEAM code.

The scheduler is responsible for choosing which Erlang process to execute at any time, similarly to an OS scheduler.

The ERTS offers automatic memory management, and keeps track of the stack and heap of each process. The garbage collectoion follows a generational copying garbage collection algorithm, extended with reference counting for some specific binaries.

BEAM is the Erlang virtual machine used for executing Erlang bytecode. It runs on an Erlang node. BEAM supports two levels of instructions: Generic instructions and Specific instructions. As there is no general Erlang Virtual Machine (EVM) definition, we will consider that BEAM and its generic instruction set are a sufficient blueprint for an EVM.

[3] For simplicity they can be thought as simple function calls

Erlang Processes

We have already discussed what Erlang processes are from a high level point of view. In practice they are just memory.

An Erlang process is basically four blocks of memory: a stack, a heap, a message area, and the Process Control Block (PCB).

The stack is used for storing return addresses, for passing arguments to functions, and for storing local variables.

The heap is used to store larger compound structures, such as tuples, lists, and maps.

The message area, is used to store the messages sent to the process from other processes. It is also known as the process mailbox.

The process control block is used to keep track of the state of the process, similarly to how the OS manages the state of an OS process.

As a result the PCB is statically allocated and contains a fixed number of fields controlling the process. The stack, heap, and mailbox on the other hand are dynamically allocated, as their size varies throughout execution.

There is also another memory area that processes have. The Process Dictionary (PD), which is a process local key-value store. As it is a small memory area, collisions are bound to happen, so each hash value points to a bucket[4]. Generally the PD is not used for saving a lot of data as its implementation is not optimal.

Scheduling

The Scheduler is responsible for choosing which Erlang process to execute at any time, similarly to an OS scheduler. A simple description is that the scheduler keeps two queues, a ready queue of processes ready to run, and a waiting queue of blocked processes that wait to receive a message. A process is moved from the waiting queue if it receives a message or if a time out happens.

The scheduler picks the first process from the ready queue and lets BEAM handle its execution for one time slice. BEAM preempts the process when the time slice is used up and adds the process back to the end of the ready queue. If the process at some point blocks in a receive during the time slice, it is moved into the waiting queue.

As we have seen a scheduler has only one executing process at any time. Thus parallelism in Erlang is achieved by simultaneously running more than one scheduler.

Garbage Collection

The Erlang Run-Time System offers automatic memory management and keeps track of the stack and heap of each process. During the execution of the program the stack and heap can both grow and shrink depending on the memory requirements. Memory management is based on a per process copying generational garbage collector, extended with reference counting for specific binaries. The reason that garbage collection (GC) happens on a per process basis is that, as described above, GC needs to block the system for a short period of time, preferable even less than a scheduling time slice, so as to retain responsiveness.

The garbage collector is initiated every time there is no free space on the heap (or the stack as they share the same allocated memory block). The garbage collector allocates a new memory area called the *to space*, similarly to other copying garbage collectors. Then it goes through the stack to find all the live roots, which it follows and copies to the new heap area located at the *to space*.

The garbage collector is generational and thus uses a heuristic to look at newer data most of the time and only occasionally check older data in order to collect them. The

[4] Buckets are implemented as lists of key-value tuples

usual smaller passes are called minor collections, where only new data is examined for collection. This is usually adequate to reclaim free space. The older part of the heap is garbage collected during the major collections or full sweeps, which are much less frequent.

Message Passing

As also described above, processes in Erlang communicate through message passing. Sending a message means that the sending process copies a message from its own memory area to the memory area of the receiving process.

In the early days of Erlang, there was no multitasking in the scheduler, so only one process could execute at a time. Thus it was safe for a sender process to just copy a message in the mailbox of a receiving process' heap.

With the introduction of multicore systems, Erlang was extended with multitasking by having several schedulers executing in parallel. As a result, it was no longer safe to write directly on another process' heap, so a main process lock would have to be acquired before any copying. However this was very slow, especially if many processes tried to send a message to the same receiver.

As a result, the concept of heap fragments outside of the main heap area, called m-bufs, was introduced. If a sender process can not acquire the main process lock of the receiver process, it instead copies the message to an m-buf. After copying finishes, the message in the m-buf is linked to the process through the mailbox, by appending it to the receiver's message queue. The garbage collector then copies the messages onto the process' heap.

In order to reduce unnecessary pressure from the GC, the mailbox is divided into two lists, one containing messages that have already been seen and one containing new (unseen) messages. This way GC does not have to check the new messages as they are still in the mailbox and they will surely survive the garbage collection. The seen messages are contained in the internal mailbox, while the unseen messages are placed in the inbox queue mailbox.

A simplified layout of all the memory areas of a process is shown in Fig. 2.3 (cite BEAM BOOK)

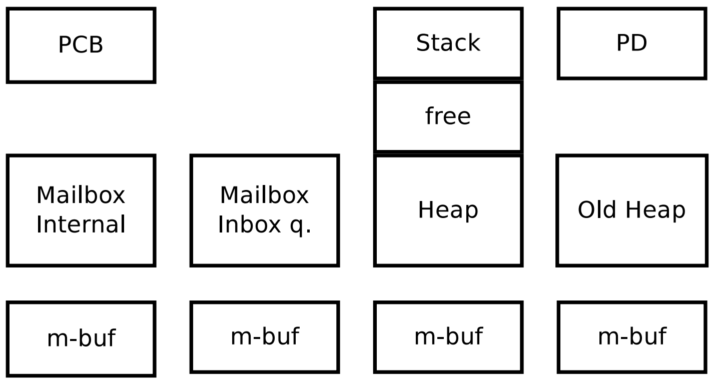

Figure 2.3: Process Memory Layout

The Erlang Compiler

The Erlang Compiler is used to generate BEAM code from Erlang source code. It contains many independent compiler passes and allows for several intermediate outputs during compilation. We will briefly describe all standard compiler passes.

1. Erlang Preprocessor epp: The first compiler pass is a tokenizer combined with a

preprocessor. First a preprocessor expands macros as tokens[5]. A macro right value should not necessarily be a valid Erlang term.

2. Parse transformations: A parse transformation allows to tweak the Erlang source language. Custom parse transforms work on an abstract syntax tree (AST) and are expected to return a new AST. They are allowed to return non valid ASTs as the validity checks happen in a later compiler pass.

3. Linter: The linter generates warnings for syntactically correct but otherwise bad code. For example, enabling the `export_all` flag leads to a warning.

4. Save AST: This pass is used to save an AST In the output file for debugging purposes. It is enabled through a compiler option. Note that the saved AST does not contain any optimization as they happen in later passes.

5. Expand: In this pass, source level language constructs are expanded to lower level Erlang constructs (e.g. Records are expanded to tuples).

6. Core Erlang: It is a strict functional intermediate language. It lies between Erlang source code and the intermediate code typically found in compilers [Carl04]. It is clean and simple, and only allows a few ways to express the same operations. Thus it facilitates code transformations and compiler optimizations. Core Erlang is regular, with well defined semantics in order to facilitate tools operating on it.

7. Kernel Erlang: It is a flat version of Core Erlang, where scoping is simpler and each variable is unique. Pattern matching in Kernel Erlang is compiled to more primitive operations.

8. BEAM Code: The last step of the standard compilation is the BEAM bytecode format.

9. Native Code: This is an optional last step of the compilation when HiPE [Joha00], the Erlang native compiler, is enabled. In this step native code is generated and stored alongside the BEAM code in the `.beam` file. This step contains many smaller steps that will be further described in Section 2.1.4.

Code Loading in Erlang

In the vast majority of programming languages changing the code that is executed in a system is a very simple and straightforward procedure. The system is stopped, the code is changed, and then the system is restarted.

However Erlang is designed for implementing real-time systems, which are often not allowed to have any downtime at all. As a result they must contain a mechanism that allows for seamless code change without the need of restarting. In a sequential system this could be easily done with a short pause, but in a system with many concurrent threads and processes the mechanism should be more elaborate. This mechanism, known as Hot Code loading, is provided by the Erlang Run-Time System and its functionality is described below.

The Erlang system allows for two versions of code of every module to be loaded in the system at the same time [Arms03]. From now on we will call those two versions *old* and *current*. The first time a module is loaded in the system, its code is considered *current*. When a new version of the module is loaded, the previously loaded code becomes *old* and the code of the new version is considered *current*.

[5] Note that the expansion is not simple string replacement as a macro will always expand as a separate token and cannot be concatenated to an existing token.

It is important to note, that in order to support hot-code loading the language makes a semantic distinction between module-local calls, which have the form `f(...)`, and so called *remote calls* which are module-qualified, i.e., have the form `m:f(...)`, and need to look up the most recently loaded version of module `m`, even from a call within `m` itself.

The benefit of having two coexisting code versions is that code change happens gradually in a process-by-process manner. Each process changes from *old* to *current* code when it executes a remote call. For clarity we present the following example:

```erlang
-module(m).
-export([loop/0]).

loop() ->
  receive
    code_switch ->
      %% A qualified function call that could lead to a code change
      m:loop();
    Otherwise ->
      %% An unqualified function call
      loop()
  end.
```

Listing 2.13: A server loop that allows for code-change

In this example, a process executing the above `loop` in an *old* code version will transition to the *current* code only if it receives the atom `code_switch` as a message.

When all processes have transitioned to *current* code, the *old* code is freed (purged) from memory. However the system is limited to two simultaneous code versions. So if a newer module version is loaded while processes still execute *old* code of this module, those processes are immediately killed and the *old* code is purged. Then the newer code becomes *current* and the previously *current* code becomes *old*. Because of that it is necessary that Erlang developers are extremely careful when loading new code, so that they do not accidentally lead to any process termination.

2.1.3 Types in Erlang

Erlang is a strongly dynamically typed language. Strongly-typed means that a term cannot be implicitly coerced to a term of another type. An example of coercion in C (a weakly-typed language) would be the use of a `char` value in place of an `int` value. In Erlang terms can only be explicitly converted to different types. Dynamically-typed means that types and type correctness are checked at runtime by added typetests. If a type error occurs, an exception is thrown. Checking types at runtime also requires a way to associate terms with types during the program execution. In Erlang this is done by combining the value of each term with a tag (a sequence of bits) that indicates its type.

Types in Erlang describe sets of terms. Predefined types exist for most standard Erlang terms, such as `integer()` and `atom()`, and for all singleton terms, such as the integer `42` and the atom `foo`. All other types are built as unions of those predefined types.

The set of Erlang types is closed under a subtyping relation $T_A \sqsubseteq T_B$ that holds if the set of terms T_B is a superset of T_A. Therefore the set of Erlang types also contains a top (`any()`) and a bottom (`none()`) type. The complete subtyping lattice [Lind05] for all predefined types is shown in Fig. 2.4.

In a type union between a type and one of its subtypes the subtype is absorbed by the supertype. For example:

```erlang
  %% The type union
  atom() | 'bar' | integer() | 42
```

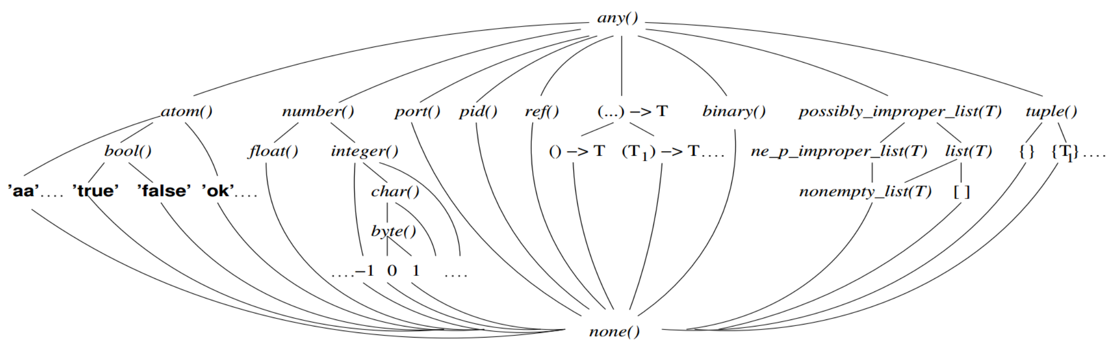

Figure 2.4: Subtyping Lattice

```

%% is the same as the following union
atom() | integer()
```

Listing 2.14: An example of a type union

Types and Function Specifications

Despite the fact that Erlang is a dynamically typed language, it is possible for the developer to explicitly declare information about the types of arguments, return values of functions, and record fields. Those type information have the following uses:

- To document function interfaces and clarify function and developer intentions.
- To facilitate static-analysis bug detection tools, such as Dialyzer [Lind04].
- To generate program documentation of various forms using documentation tools, such as EDoc [Eric].

Documentation of functions is usually written in program comments. However comments often tend to drift out of phase, and lag behind the code itself. This is the main reason to use function specifications which, when combined with a bug detection tool that checks if the specifications match the implementation, will not drift out of phase.

The Erlang Tagging Scheme

As also stated above, tagging means that in the memory representation of a term, some bits are reserved for a type tag. Erlang terms are separated into immediates and boxed terms. Immediates can fit in a machine word, whereas boxed terms consist of two parts, a tagged pointer and a number of words stored on the process heap. The words stored in the heap contain a header and a body.

The basic idea is that the least significant bits of a word are used as tags. As most modern CPU architectures align 32 or 64 bit words, pointers have at least 2 unused bits. Those are used as tag but they are not enough to represent all Erlang types. Therefore more bits are used as needed.

Success Typings

A constraint-based type inference algorithm based on success typings [Lind06] is used for Erlang in order to find possible bugs [Lind04], to annotate programs with function specifications [Lind05], and to perform optimizations [Joha00]. The main goal of this type inference is to uncover as much of the type information that is present without ever underestimating a type. A definition for success typings follows below [Lind06]:

DEFINITION 1 (Success Typings). *A success typing of a function f is a type signature, $(\bar{\alpha}) \rightarrow \beta$, such that whenever an application $f(\bar{p})$ reduces to a value v, then $v \in \beta$ and $\bar{p} \in \bar{\alpha}$.*

So the essence of a success typing $\bar{\alpha} \rightarrow \beta$ for a function f is that if the arguments of the function are in its domain, the application might succeed, but if they are not the function call will definitely fail.

An important point that needs to be mentioned is that this type system does not support parametric polymorphism and therefore does not capture any information about the relation of its arguments to its result. An example is shown below:

```erlang
%% A simple function that tags its argument
tag_1(N) when is_atom(N) -> {atom, N};
tag_1(N) when is_float(N) -> {float, N};
tag_1(N) when is_integer(N) -> {integer, N}.

%% Its inferred type would be
tag_1/1 :: (atom() | number()) -> {'atom', atom()}
                                | {'float', float()}
                                | {'integer', integer()}.
```

Listing 2.15: An example of a success typing that misses the relation of the arguments to the result.

In this example the fact the the input argument is the same as the second element of the output tuple is lost.

2.1.4 HiPE

HiPE is a native code compiler for Erlang. It offers flexible integration between emulated and native code. Its main goal is to allow compiling specific parts of the application, while keeping others emulated in order to achieve the best balance between performance and portability.

Before the development of HiPE all implementations of Erlang were based on emulators of virtual machines. This improved portability but incurred performance penalties. Also byte-code emulators usually result in smaller object code size than C based or native code compilers. While in many areas, object code size becomes less and less of a problem, there are areas (such as embedded software) where it is still a potential problem.

An overview of an Erlang/OTP system extended with HiPE is shown in Fig. 2.5 [Sago03].

The smallest unit of compilation in HiPE is a single function. However it produces higher quality of native code when it compiles whole modules, as it performs global analyses that greatly benefit all optimizations.

Intermediate Representations

HiPE has four intermediate representations, [Joha00].

- An internal representation of BEAM bytecode.

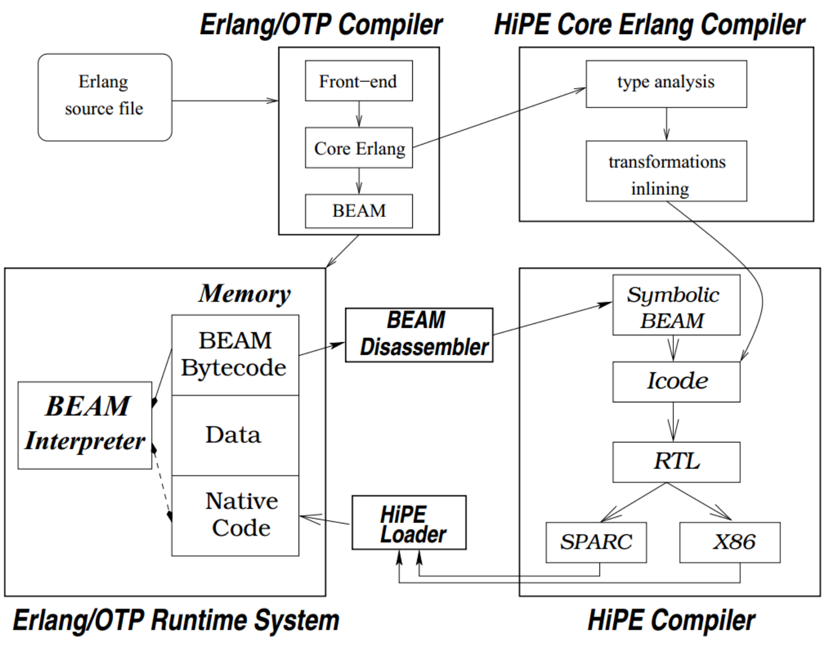

Figure 2.5: Structure of a HiPE enabled Erlang/OTP system

- A higher level intermediate language, which is called ICode.
- A general register transfer language, which is called RTL.
- A machine specific assembly language for several architectures (such as SPARC, x86, etc).

ICode, RTL, and the assembly languages are all internally represented as control flow graphs of basic blocks.

ICode is based on a register-oriented virtual machine for Erlang. It supports an infinite number of registers which are used to store arguments and temporaries. All values in ICode are proper Erlang terms. In addition, the call stack is implicit and preserves registers. Finally, as ICode is a high level intermediate language, bookkeeping operations (such as heap overflow checks, context switching, etc) are implicit.

Several standard optimizations are applied to ICode, such as copy and constant propagation and constant folding. Dead code removal is then performed to remove assignments to dead temporaries.

Then ICode is translated to RTL, and during this process unreachable code is removed as only reachable basic blocks are translated to RTL.

Call stack management and the saving and restoring of registers around calls are made explicit in RTL. Then the standard optimizations that were applied in ICode are applied again. Also heap overflow tests are made explicit and are propagated backwards as far as possible, in order to be merged together to limit performance overhead. Registers in RTL are separated in tagged and untagged, where the untagged registers can hold plain values to optimize performance. However, untagged registers cannot live across function calls.

The RTL code is then translated to assembly code and registers are assigned using a graph coloring register allocator similar to the one in Briggs et al. [Brig94]. Finally symbolic references to atoms and functions are replaced by their real values in the running system, memory is allocated for the code, and then code is linked to the system.

Splitting compilation in three separate intermediate representations conceptually provides nice abstraction levels. It simplifies optimization experimentation and the development of the compiler. However many basic optimization such as constant propagation and folding happen in all IRs, potentially slowing down the compilation [Joha03].

The HiPE Linker

As described in Section 2.1.2, Erlang allows upgrading code at runtime, without stopping any process executing the old version of the code.

The BEAM system handles this by maintaining a global table of all loaded modules. For each module it keeps the name, the list of exported functions, and the location of the current and precious code segments. The exported functions always refer to the current code segments. A remote (qualified) function call leads to a lookup of the module and function name. If the module is loaded and the function is found in its exported list, the emulator starts executing it. If not an error handler is invoked.

In contrast to BEAM, in native code a function call is just a direct machine level jump to an absolute address. When the caller's code is linked, the linker initialises the call to directly jump to the callee. If the callee has not been loaded the call will be directed to a stub containing the appropriate error handling. If the callee is loaded but as BEAM bytecode, then the call is again directed to a stub which invokes BEAM to execute the callee's bytecode.

In order for HiPE to support hot code loading, information about all call sites in native code has to be maintained in some data structure. Every time a module is loaded or unloaded, native call sites referring to that module are updated based on the kept information.

When a module is updated with a new version of emulated code, all remote function calls from native code to that module are located and patched to call the new emulated code, via new native-to-emulated code stubs.

When an emulated function is compiled to native and then loaded to the system, each call to it from native code is updated with its new address. The preciously existing native-to-emulated code stub is deallocated.

When a module is unloaded, all native code call sites that refer to that module are patched to invoke an error handling stub. Also every native code call site existing in this module is removed from the linker's data structure, so as to not be updated in the future.

HiPE's dual stack approach

It is interesting to note that HiPE uses two stacks for each process, an emulated code stack (estack), which is the standard one, and a native code stack (nstack).

A disadvantage of having two stacks is that exception handling is handled by creating a linked list of catch frames on the stack, so there might be pointers from one stack to the other. Because of that, relocation of one of the two stacks, which is done by the garbage collector to increase its size, means that every catch frame pointer on the other stack that points to the relocated one must be updated.

An advantage of this approach is that the garbage collector can easily deal with the different stack frame layouts in native and emulated mode. With a single stack, the garbage collecting code would have to switch modes while scanning to be able to handle the different stack frame layouts.

Mode Switches

A mode-switch is when control transitions from native code to emulated code or vice versa. HiPE was designed in a way so that no runtime overhead occurs if there are no mode-switches.

Mode-switches occur in function calls and returns between a native and an emulated function. Also they occur when errors are thrown in code executing in one mode, and catched in code that executes in the other mode.

HiPE has to detect those mode-switches without heavy runtime checks so as to not encumber performance. We will describe how HiPE detects mode-switches in case of calls, returns, and throws.

In case of calls, a call from native to emulated code passes from a native-to-emulated stub, which causes a switch to emulated mode. In contrast all native compiled functions' emulated code begins with a special emulator instruction that causes a switch to native mode and then jump to the native code itself.

In case of returns, when a call causes a mode-switch, a new "continuation" call frame is created in callee's mode. This frame is saved in the mode's stack and contains a return address to code which causes a switch back to the caller's mode. When returning from native to emulated code, the return address points to machine code in the runtime system. For returns from emulated to native code, the return address points to a special emulator instruction that switches mode to native. This design decision makes sure that no overhead is caused while the mode stays the same, and that performance cost only incurs when there is a mode-switch.

Mode-switches in throws are handled similarly to mode-switches in function returns. When a call causes a mode switch, a new exception catch frame is created in the mode of the callee. The handler address points to code that switches back to the caller's mode and then re-throws the exception.

The method based on stack frames that is described above is efficient and easy to implement. However, it violates the tail-call optimization that happens in tail-recursive functions and is a fundamental Erlang feature. Consider a native code function that tail-calls an emulated code function that tail-calls the same native code function and so on. In this scenario, each call introduces a new call stack-frame to the stack, so stack space will grow and eventually fill up all memory.

HiPE deals with this with a simple check when a tail-call mode-switch happens.

1. If the current return frame is a mode-switch frame (that means that the function that called the current function was in a different mode) then:
   (a) Remove the mode-switch return frame from the caller's stack.
   (b) Remove the mode-switch catch frame from the caller's stack.
   (c) Normally invoke the callee. (This way no extra mode-switch frame is inserted in the stack.)
2. Otherwise:
   (a) Push a mode-switch catch frame on the callee's stack.
   (b) Push a mode-switch return frame on the callee's stack.
   (c) Normally invoke the callee.

Using this method adjacent mode-switches are never created so proper tail-recursive behaviour is maintained. Then above test is only executed when a mode-switch tail-call happens so it does not hinder performance.

Some special care should also be taken in case of suspending and resuming process execution, as ERTS handles the scheduling of processes. So when a process in native mode is suspended and an emulated one is resumed, a mode-switch effectively happens. The way HiPE handles it is that when a process is created or resumed, it is always assumed that it executes in emulated mode. When a process is suspended while executing in native code, HiPE sets the resume address in the PCB to point to a special emulator instruction that transfers control to native mode and then resumes the execution normally.

Based on the above, we can conclude that mode-switches, especially when happening a lot, are relatively expensive. Because of that Erlang application developers need to be careful to not create lengthy call chains of functions in different modes.

Optimizations: Type Propagation

Erlang is a dynamically typed language and this generally allows for faster prototyping and experimentation in the early development stages. However this also leads to the introduction of many type tests during the code execution to make sure that the operations are performed on meaningful data types. An example of an operation performed on meaningless types could be a division of a float by a list.

HiPE developers initially tried to tacke this by implementing a local type propagation pass, which discovers type information (on a per function basis) and propagates that information to eliminate redundant type tests [Sago03]. That type information is also used to transform some polymorphic general operations to faster specialized ones.

However, by not knowing anything about the function arguments, only minimal type information can be discovered. Because of that, HiPE also has a global type analyzer which processes whole modules. The analyzer is able to deduce more specific type information especially when the module is relatively closed to the outer world, which means that not many functions are exported out of it. This allows HiPE to make stricter assumptions about the arguments of functions that are not exported, which then lead to more type tests being eliminated. Type information also helps avoiding repeatedly tagging and untagging values between operations.

It is important to note that the type analyzer is not a type checker, and does not produce any warnings for ill-defined functions. Also the user cannot interact with it in any way.

Optimizations: Float Handling

Atomic Erlang values are represented as tagged 32 or 64 bit words in the runtime system. If a tagged value is too big to fit into one machine word, the value is boxed and saved on heap. Because of that, floating point numbers are typically boxed, so every time a floating point operation must be done, the arguments are unboxed and then the result is boxed back.

To avoid this overhead BEAM supports special floating point instructions that operate directly on untagged values, so many boxing/unboxing operations are avoided. However BEAM code is interpreted so floating point arithmetic is not taking advantage of the FPU of the target architecture.

HiPE extends on BEAM by mapping floating point values to the FPU and keeping them there for as long as possible, reducing the overhead of floating point operations. In combination with the type propagation that has ben described above, HiPE can very efficiently handle floating point operations with the minimum number of type tests.

2.2 JIT Compilation

Managed languages, such as Java [Arno00a] and C# [Hejl06] support the “compile-once, run-anywhere” model for code generation and distribution. This allows the generation of programs that are portable and can be executed on any device equipped with the corresponding language virtual machine. Because of that the format of the generated program has to be independent from a specific native architecture and thus those languages are often interpreted before program execution. This in turn leads to performance overhead compared to native generated code. This trade-off between native compiled code and interpreted code has led researchers to explore just in time (JIT) compilation (compilation during the program execution) which is a combination of the two.

There are two main JIT compiler categories based on the basic compilation block, function JIT compilers and tracing JIT compilers.

2.2.1 Function JIT Compilation

Function JIT compilers profile functions during execution and then compile and optimize them based on some decision algorithm. The first JIT compilers compiled a function and applied a fixed set of optimizations the first time it was called [Deut84, Cham91]. However this proved inefficient performance wise, as there were many execution pauses to compile each function. That is why most function JIT compilers nowadays employ a staged emulation model [Hans74], where each function is initially interpreted and later on optimized by the JIT compiler if the need arises.

An essential feature of function JIT Compilers is selective compilation, which is based on the observation that most programs spend a large majority of their execution time in a small portion of the code. Selective compilation uses runtime profiling to determine which functions take up the biggest part of execution time, also called “hot” functions, in order to only compile and optimize those [Holz94, Suga00, Cier00] and limit the JIT overhead while getting a pretty big performance benefit.

Unfortunately finding the hot functions to compile requires future program execution information and cannot be predicted, so most JIT strategies make the assumption that current hot functions will remain hot in the future. The most popular profiling approaches are based on some form of counter, either counting the number of function calls [Hans74, Kotz08] or by periodically interrupting execution, checking which function is executing at that time [Arno05] and updating a counter. Both methods consider a function to be hot when its counter has exceeded a specified threshold. Choosing that threshold value is an important task because setting it too low could lead to very aggressive compiling which will degrade execution performance, while setting it too high could lead to a very conservative JIT compiler which does not optimize any methods at all. Most JIT Compiler designers determine that threshold value by measuring the JIT compiler performance with various threshold values over a large benchmark suite.

A very important characteristic of function JIT compilers is the ability to use runtime information to tailor optimizations to a specific execution [Kist03, Burg97, Burg98, Gran99], which has been shown to yield substantial performance improvements.

One of those essential optimizations was developed for the first generation of the Self JIT Compiler and is called customization [Cham89]. The main idea was that instead of dynamically compiling a function into native code that works for any invocation, the JIT Compiler would produce a specialized version of the function based on the particular context. Type information is the biggest source of runtime information that was gathered and used to produce specialized versions of functions. This optimization was especially beneficial for Self, where every operation is dynamic and changeable so no static information is available to the compiler [Ayco03].

Profile-directed inlining using runtime profile data has been widely studied [AdlT03, Cier00, Haze03] and has been incorporated in many JIT compilers. That is because statically predicting the effect of inlining a call site is a computationally expensive task. However by having access to runtime information such as the execution time spent in each function and the number of times every call site has been invoked, inlining could be driven to provide the same performance benefits with much simpler decision algorithms. It has been shown that profile-driven inlining decision mechanisms outperform ones that only rely on static data [Arno02, Suga02].

2.2.2 Tracing JIT Compilation

Tracing Just-In-Time compilers (TJITs) determine frequently executed traces (hot paths) in running programs and emit optimized machine code for these traces. The TJIT observes execution until a "hot" instruction sequence (trace) is identified. Then the TJIT generates an optimized version of the trace. Subsequent encounters of the "hot" trace's entry address during interpretation lead to jump to the top of the optimized version of the trace. When control exits the optimized trace, the TJIT continues interpreting normally.

It is important to understand how a TJIT selects which traces to optimize. The main reasoning is that a TJIT aims more for predictability and less for accuracy. So for example, a trace that is hot for a short period of time, but overall does not contribute to the execution time of the program a lot, may still be an important trace to identify.

In order to understand how a standard TJIT works in detail, we will describe the functionality of Dynamo [Bala00] which was one of the first TJIT compilers. The TJIT maintains a counter for some specific program points, called start-of-trace, such as the target addresses of backward taken branches which are very likely to be loop headers. Each counter is increased when execution passes from the target address. If a counter exceeds a preset threshold, the TJIT records all instructions starting from the address associated with the counter. The recording stops when some specific conditions are met and the recorded trace is saved to later be optimized. This trace selection scheme is called most recently executed tail (MRET) and the insight behind it is that when an instruction becomes "hot", it is likely that the very next sequence of instructions is also going to be "hot". Thus instead of recording all the branches after the start-of-trace, which would lead in a big memory overhead, it just records the sequential tail of instructions following the start-of-trace in that particular execution.

It is important to note that being sequential the trace represents only one of the many possible paths throughout the code. To ensure correctness, the trace contains a guard at every possible point where the path could have followed another direction (e.g. in conditions).

In contrast to Dynamo, Gal et al. [Gal09] have decided to extend the simple sequential tracing with the notion of trace trees, which supposedly offer performance benefits as they better model simple control flow in execution paths.

In case of PyPy, Bolz et al. [Bolz09] designed a TJIT that instead of tracing the program, traces the interpreter itself, with the goal of extending PyPy as the TJIT of many dynamic languages by implementing their interpreter in RPython[6].

Tracing JITs have a number of advantages over static compilers in that they are able to harness runtime information, such as indirect jump and call targets [Bebe10]. In addition, statically compiling dynamic languages, such as Javascript, to efficient code ahead of runtime is difficult due to the dynamic nature of the operations in the language. When statically compiling code in those language, no assumptions can be made about the arguments of an operation, so many runtime checks have to be implemented. TJITS are also considered to be able to more aggressively optimize loops by unrolling as they focus on specific hot traces.

[6] A subset of Python that PyPy supports.

2.3 Other JiT compilers for Erlang

JiT compilation has been investigated in the context of Erlang several times in the past, both distant and more recent. For example, both Jerico [Joha96], the first native code compiler for Erlang (based on JAM) circa 1996, and early versions of HiPE contained some support for JiT compilation. However, this support never became robust to the point that it could be used in production.

More recently, two attempts to develop tracing JiTs for Erlang have been made, namely BEAMJIT [Drej14] and Pyrlang [Huan16].

2.3.1 BEAMJIT

BEAMJIT [Drej14] is a tracing Just-in-Time compiling runtime for Erlang. BEAMJIT uses a tracing strategy for deciding which code sequences to compile to native code and the LLVM toolkit [Latt04] for optimization and native code emission. It extends the base BEAM implementation with support for profiling, tracing, native code compilation, and support for switching between these three (profiling, tracing, and native) execution modes.

Performance-wise, back in 2014, BEAMJIT reportedly managed to reduce the execution time of some small Erlang benchmarks by 25–40% compared to BEAM, but there were also many other benchmarks where it performed worse than BEAM [Drej14]. Moreover, the same paper reported that "HiPE provides such a large performance improvement compared to plain BEAM that a comparison to BEAMJIT would be uninteresting" [Drej14, Sect. 5]. At the time of this writing (May 2018), BEAMJIT is not yet a complete implementation of Erlang; for example, it does not yet support floats. Although work is ongoing in extending BEAMJIT, its overall performance, which has improved, does not yet surpass that of HiPE.[7] Since BEAMJIT is not available, not even as a binary, we cannot compare against it.

2.3.2 Pyrlang

The second attempt, Pyrlang [Huan16], is an alternative virtual machine for the BEAM byte code which uses RPython's meta-tracing JiT compiler [Bolz09] as a backend in order to improve the sequential performance of Erlang programs. Meta-tracing JiT compilers are tracing JiT compilers that trace and optimize an interpreter instead of the program itself. The same idea has been previously applied to Racket, in the form of the Pycket [Baum15] tracing JiT. The Pyrlang paper [Huan16] reports that Pyrlang achieves average performance which is 38.3% faster than BEAM and 25.2% slower than HiPE, on a suite of sequential benchmarks. Currently, Pyrlang is a research prototype and not yet a complete implementation of Erlang, even for the sequential part of the language. On the other hand, unlike BEAMJIT, Pyrlang is available, and we will directly compare against it in Chapter 5.

[7] Lukas Larsson, private communication, May 2018.

Chapter 3

HiPErJiT Architecture and Components

The functionality of HiPErJiT can be briefly described as follows. It profiles executed modules, maintaining runtime data such as execution time and call graphs. It then decides which modules to compile and optimize based on the collected data. Finally, it compiles and loads the JiT-compiled modules in the runtime system. Each of these tasks is handled by a separate component.

Controller The central controlling unit which decides which modules should be profiled and which should be optimized based on runtime data.

Profiler An intermediate layer between the controller and the low-level Erlang profilers. It gathers profiling information, cleans up and organizes it, and sends it back to the controller for further processing.

Compiler+Loader An intermediate layer between the controller and HiPE. It compiles the modules chosen by the controller and then loads them into the runtime system.

The architecture of the HiPErJiT compiler can be seen in Fig. 3.1.

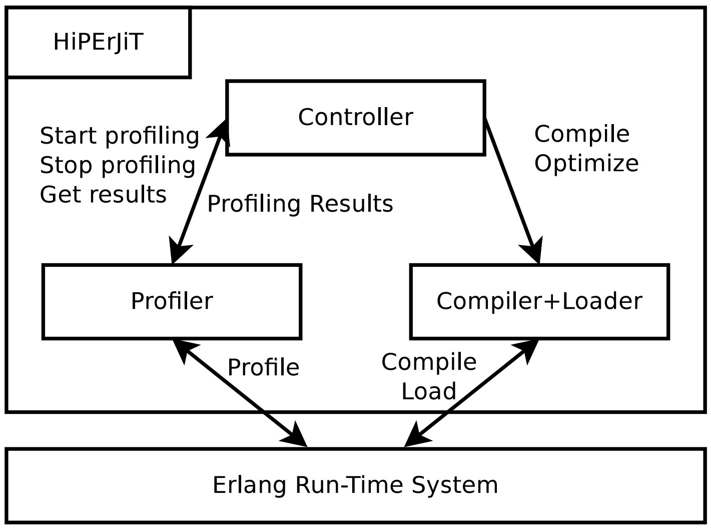

Figure 3.1: High level architecture of the JIT Compiler

3.1 Controller

The controller, as stated above, is the fundamental module of HiPErJiT. It chooses the modules to profile, and uses the profiling data to decide which modules to compile and optimize. It is essential that no user input is needed to drive decision making. Our design extends a lot of concepts from the work on the Jalapeño JVM [Arno00b].

Traditionally, many JiT compilers use a lightweight call frequency method to drive compilation [Ayco03]. This method maintains a counter for each function and increments it every time the function is called. When the counter surpasses a specified threshold, JiT compilation is triggered. This approach, while having very low overhead, does not give precise information about the program execution.

Instead, HiPErJiT makes its decision based on a simple cost-benefit analysis. Compilation for a module is triggered when its predicted future execution time when compiled, combined with its compilation time, would be less than its future execution time when interpreted:

$$FutureExecTime_c + CompTime < FutureExecTime_i$$

Of course, it is not possible to predict the future execution time of a module or the time needed to compile it, so some estimations need to be made. First of all, we need to estimate future execution time. The results of a study about the lifetime of UNIX processes [Harc97] show that the total execution time of UNIX processes follows a Pareto (heavy-tailed) distribution. The mean remaining waiting time of this distribution is analogous to the amount of time that has passed already. Thus, assuming that the analogy between a module and a UNIX process holds, we consider future execution time of a module to be equal to its execution time until now $FutureExecTime = ExecTime$. In addition, we consider that compiled code has a constant relative speedup to the interpreted code, thus $ExecTime_c * Speedup_c = ExecTime_i$. Finally, we consider that the compilation time for a module depends linearly on its size, so $CompTime = C * Size$. In short, the condition to check is:

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

If this condition holds the module is “worth” compiling.

We conducted two experiments in order to find suitable values for the condition parameters $Speedup_c$ and C. In the first one, we executed a set of Erlang applications before and after compiling their modules to native code. The average speedup we measured was 2 and we used it as an estimate for the $Speedup_c$ parameter. In the second experiment, we compiled a set of modules of varying sizes, measured their compilation time, and fitted a line to the set of given points using the Least Squares method (cf. Fig. 3.2).

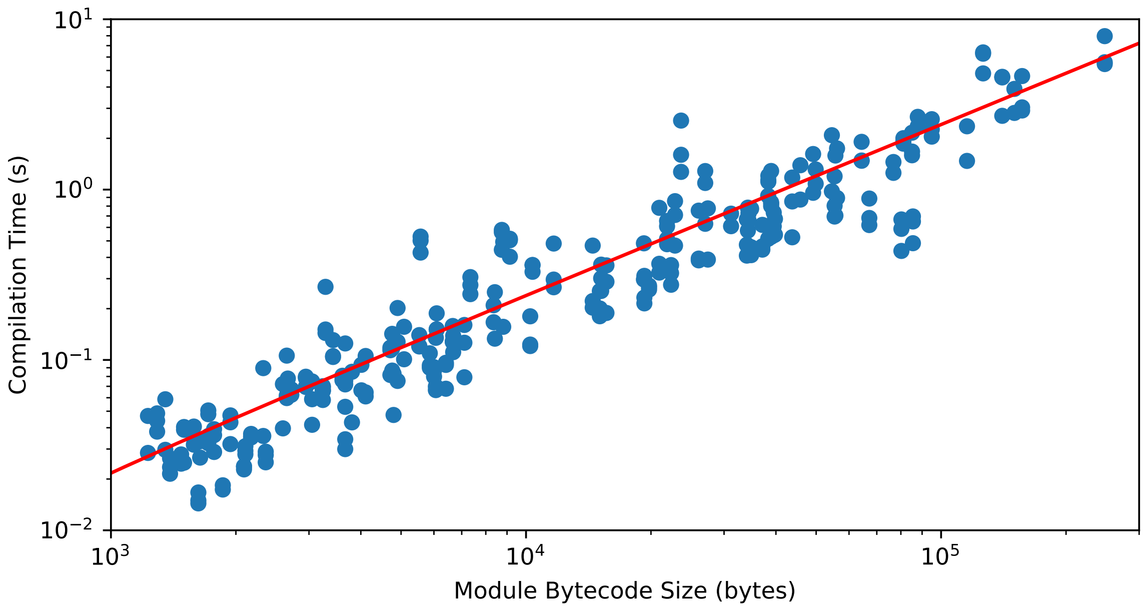

Figure 3.2: Compilation Times of Modules related to their Bytecode Size

The line has a slope of $2.5e^{-5}$ so that is also the estimated compilation cost per byte of byte code. It is worth mentioning that, in reality, the compilation time does not depend only on module size, but on many other factors (e.g., branching, exported functions, etc). However, we consider this estimate to be adequate for our goal.

Finally, HiPErJiT uses a feedback-directed scheme to improve the precision of the compilation decision condition when possible. HiPErJiT measures the compilation time of each module it compiles to native code and stores it in a persistent key-value storage, where the key is a pair of the module name and the MD5 hash value of its byte code. If the same version of that module is considered for compilation at a subsequent time, HiPErJiT will use the stored compilation time measurement in place of $CompTime$.

3.2 Profiler

The profiler is responsible for efficiently profiling executed code. Its architecture, which can be seen in Fig. 3.3, consists of:

- The profiler core, which receives the profiling commands from the controller and transfers them to ERTS. It also receives the profiling results from the individual profilers and transfers them back to the controller.
- The router, which receives all profiling messages from ERTS and routes them to the correct individual profiler.
- The individual profilers, which handle profiling messages, aggregate them, and transfer the results to the profiler core. Each individual profiler handles a different subset of the execution data, namely function calls, execution time, argument types, and process lifetime.

We designed the profiler in a way that facilitates the addition and removal of individual profilers.

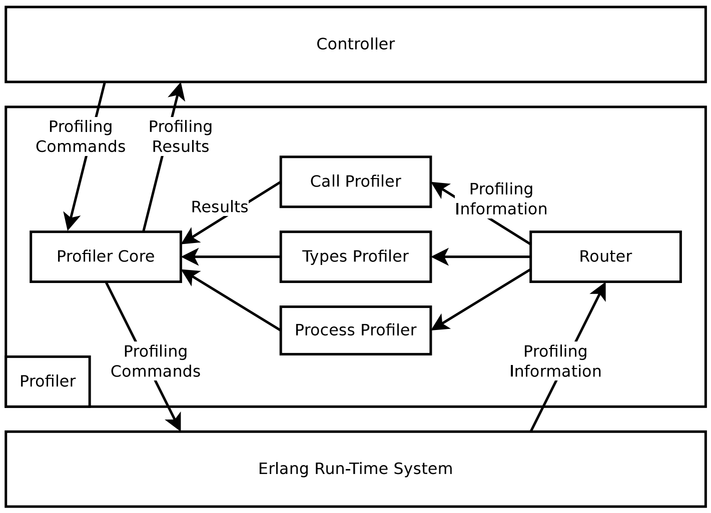

Figure 3.3: Profiler Architecture and Internal Components

3.2.1 Profiling Execution Time

We initially used the execution time tracing functionality of the ERTS, in order to profile execution time. However, due to very high overhead we decided to profile execution time using a different method, by tracing all function calls, function returns, and process scheduling actions. To better explain our method, an example execution period of two schedulers is shown in Fig. 3.4.

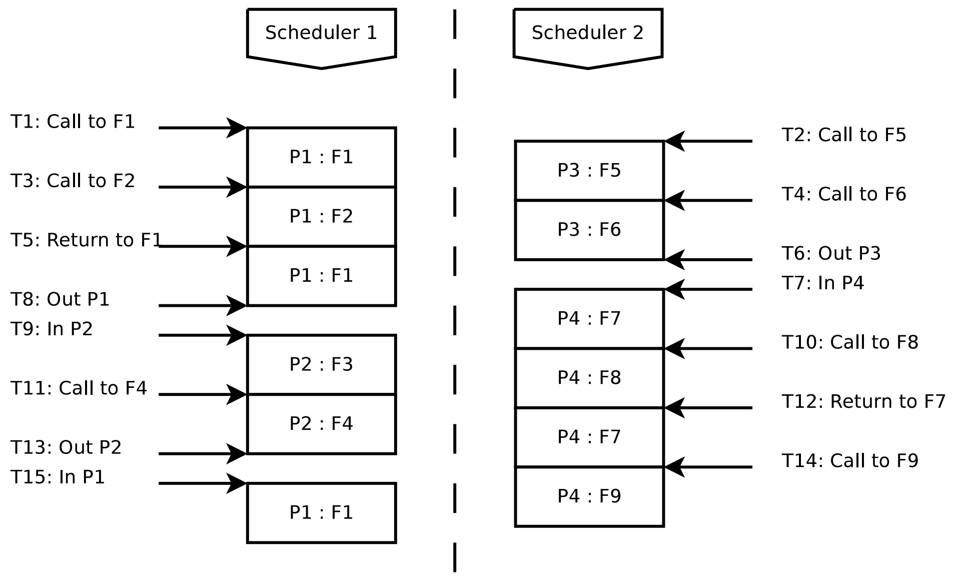

Figure 3.4: The trace messages sent during an execution period of two schedulers

Each rectangle depicts a time slice in the scheduler and it is identified by the process that is running and the function that it executes. Tis are timestamps, Pis are processes, and Fis are functions. In and Out mean that a process has been scheduled in or out at that time. Each arrow represents a trace message that the profiler receives from ERTS. Messages have the form $\langle Action, Timestamp, Process, Function \rangle$, where $Action$ can be any of the following values: $call$, $return\_to$, $sched\_in$, $sched\_out$.

For each sequence of trace messages that refer to the same process P, i.e., a sequence of form $[(sched\_in, T_1, P, F_1), (A_2, T_2, P, F_2), \ldots, (A_{n-1}, T_{n-1}, P, F_{n-1}), (sched\_out, T_n, P, F_n)]$, we can compute the time spent at each function by finding the difference of the timestamps in each pair of adjacent messages, so $[(T_2 - T_1, F_1), (T_3 - T_2, F_2), ..., (T_n - T_{n-1}, F_{n-1})]$. The sum of all the differences for each function gives the total execution time spent in each function.

Our method put less overhead on total performance and gave equally precise results. Also it allowed us to track the number of calls between a pair of functions which is later used for profile driven inlining (Section 4.2). However, in concurrent programs, the rate at which messages are sent to the profiler can increase uncontrollably, thus flooding its mailbox, leading to high memory consumption and lack of responsiveness.

This problem is actually very similar to the responsiveness of data streaming systems, and has been previously studied by many [Reis05, Babc04]. Data streaming systems often have to deal with bursts of data that cannot be handled by available system resources.This is most commonly tackled by a form of adaptive load shedding that drops data before it enters the system.

Based on the above, we decided to drop trace messages if the load is too high in order to keep responsiveness at a stable level. We achieved this by borrowing the notion of reductions from the Erlang scheduler. We extended the tracer so that every R reductions, it would check the whole mailbox to find a results request. If not found it would reset its reduction counter and continue handling trace messages. Using this technique we were able to have a "responsive" tracer even when more trace messages than it could handle were sent to it. All the remaining unhandled messages in the mailbox are dropped. This obviously leads to information loss, but this should not discourage us, as in the end of the day the JIT compiler needs the execution time tracing results to decide whether to compile each module and thus extreme precision is valued less than performance overhead.

After those improvements the tracing overhead was lower than the standard execution time tracing mechanism, the tracing results were adequately precise, and the system was responsive for the "mostly sequential" testcases. However this did not hold for most complicated multi-process testcases, and this lead us to the following realization.

Our system has a very important difference with the data streaming systems that are studied in the bibliography. Its difference is that in our case, the data producer is part of the system, and that it also puts overhead on system performance. Thus, while load shedding makes our system more responsive, it does not reduce the overhead imposed by the data producer, which still creates and sends trace messages that are eventually going to be dropped. As a result, we slightly altered our mechanism so that it stops the producer completely, instead of just dropping its messages. We inserted a probing mechanism that checks whether the mailbox exceeds some maximum size (which means that more trace messages that can be handled are sent to our system) and if so stops the tracing messages from being sent completely.

This alteration offered great performance benefits, both in execution time and memory usage. The execution time benefit of the systems discussed in this section is shown in Fig. 3.5. Note that the execution times for custom time tracing in dialyzer_plt and orbit_par_seq are not shown because those executions depleted the available memory and crashed. This is mainly because those benchmarks spawn many processes and then overflow the tracer with tracing messages.

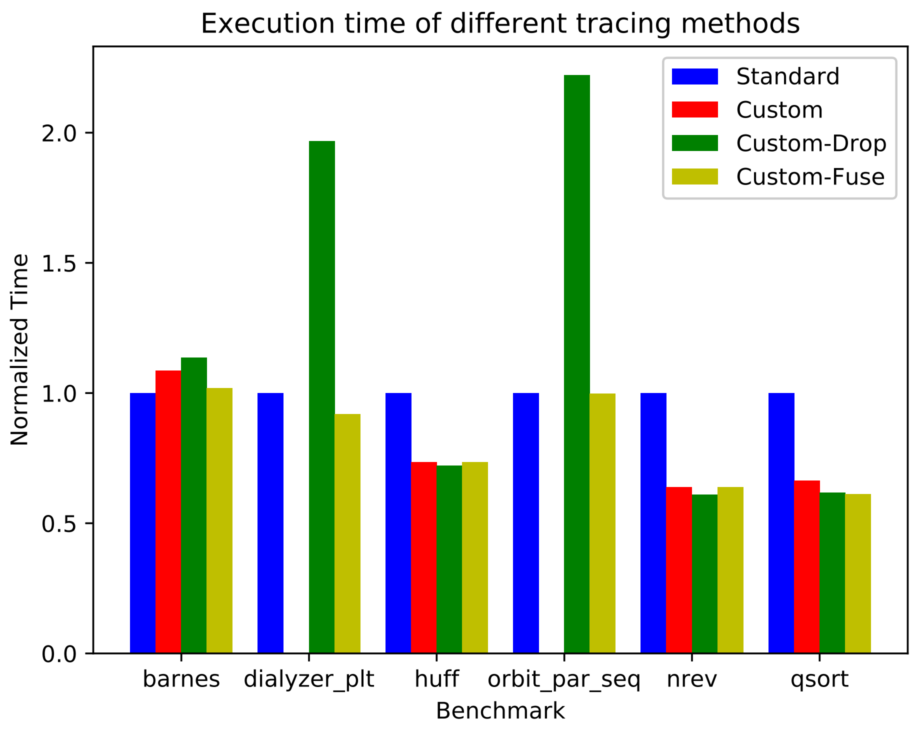

Figure 3.5: The execution time of some benchmarks with different tracing methods enabled

The tracing results of our method are similar (but not equal) to the ones acquired from the standard time tracing mechanism. Small differences can be attributed to the following reasons:

- It is not possible to trace both local and external calls with the trace mechanism, so only local calls are traced. This leads to imprecisions, especially in cases of large external call chains as our method considers that all that time is spent in the latest local called function.

- The other source of imprecisions is that tracing pauses periodically so some scheduling actions are not traced, thus leading to miscalculations of the execution state. This is especially evident when the number of processes is much larger than the number of schedulers.

The reduction of the tracing overhead by our mechanism justifies using it in the context of HiPErJiT despite the slight imprecisions in the tracing results.

As future work we plan to refine our method so that it becomes as stable and precise as the standard one. In order to achieve that, we plan to extend it to simultaneously trace both local and global calls. In addition we will work on the precision of its results by making sure that the scheduling state is consistent when pausing and restarting.

3.2.2 Type Tracing

The profiler is also responsible for type tracing. As we will see, HiPErJiT contains a compiler pass that uses type information for the arguments of functions in order to create type-specialized, and hopefully better performing, versions of functions. This type information is extracted from type specifications in Erlang source code (if they exist) and from runtime type information which is collected and aggregated as described below.

For modules that the profiler has selected for profiling, the arguments of function calls to this module are also profiled. The ERTS low-level profiling functionality returns the complete argument values, so in principle their types can be determined. However, functions could be called with complicated data structures as arguments, and determining their types precisely requires a complete traversal. As this could lead to a significant performance overhead, the profiler approximates their types by a limited-depth term traversal. In other words, *depth-k type abstraction* is used. Every type T is represented as a tree with leaves that are either singleton types or type constructors with zero arguments, and internal nodes that are type constructors with one or more arguments. The depth of each node is defined as its distance from the root. A depth-k type T_k is a tree where every node with depth $\geq k$ is pruned and over-approximated with the top type (`any()`). For example, in Erlang's type language [Lind05], which supports unions of singleton types, the Erlang term `{foo,{bar,[{a,17},{a,42}]}}` has type `{'foo',{'bar',list({'a',17|42})}}`, where `'A'` denotes the singleton atom type `A`. Its depth-1 and depth-2 type abstractions are `{'foo',{any(),any()}}` and `{'foo',{'bar',list(any())}}`.

The following two properties should hold for depth-k type abstractions:

1. $\forall k.k \geq 0 \Rightarrow T \sqsubseteq T_k$ where $\sqsubseteq$ is the subtype relation.

2. $\forall t.t \neq T \Rightarrow \exists k.\forall i.i \geq k \Rightarrow t \not\sqsubseteq T_i$

The first property guarantees correctness while the second allows us to improve the approximation precision by choosing a greater k, thus allowing us to trade performance for precision or vice versa.

Another problem that we faced is that Erlang collections can contain elements of different types. Traversing them in order to find their complete type could also create undesired overhead. As a result, we decided to optimistically estimate the collection element types. Although Erlang collections can contain elements of many different types, this is rarely the case in practice. In most programs, collections usually contain elements of the same type. This, combined with the fact that HiPErJiT uses the runtime type information to specialize some functions for specific type arguments, gives us the opportunity to be more optimistic while deducing the types of values. Therefore, we decided to approximate the types of Erlang collections (currently lists and maps) by considering only a small subset of their elements.

Considering all of the above, our type tracer is able to estimate the types of Erlang values in an efficient way. One point worth mentioning is that initially, we had implemented this functionality based on the standard Erlang trace mechanism that traces the arguments and return values of each function call. This had a negative impact on performance because, as also stated in Section 2.1.1, messages in Erlang are copied, so in fact all function call arguments and return values were copied as the program was being executed, leading to detrimental memory and execution time overheads. We addressed this issue by implementing this functionality as an Erlang NIF before the trace message sending, thus preventing unnecessary copying of values that were not needed.

What is left is a way to generalize and aggregate the type information acquired through the profiling of many function calls. As described in Section 2.1.3, types in Erlang are internally represented using a subtyping system, thus forming a type lattice. Because of that, there exists a supremum operation that can be used to aggregate type information that has been acquired through different traces.

3.2.3 Profiling Processes

Significant effort has been made to ensure that the overhead of HiPErJiT does not increase with the number of concurrent processes. Initially, performance was mediocre when executing concurrent applications with a large number ($\gg 100$) of spawned processes because of profiling overhead. While profiling a process, every function call triggers an action that sends a message to the profiler. The problem arises when many processes execute concurrently, where a lot of execution time is needed by the profiler process to handle all the messages being sent to it. In addition, the memory consumption of the profiler skyrocketed as messages arrived with a higher rate than they were consumed.

To avoid such problems, HiPErJiT employs *probabilistic profiling*. The idea is based on the following observation. Massively concurrent applications usually consist of a lot of sibling processes that have identical functionality. Because of that, it is reasonable to sample a part of the hundreds (or even thousands) of processes and only profile them to get information about the system as a whole.

This sample should not be taken at random, but it should rather follow the principle described above. We maintain a process tree by monitoring the lifetime of all processes. The nodes of the process tree are of the following type:

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

The `pid()` is the process identifier, the `mfa()` is the initial function of the process, and the `[ptnode()]` is a list of its children processes, which is empty if it is a leaf. The essence of the process tree is that sibling subtrees which share the same structure and execute the same initial function usually have the same functionality. Thus we could only profile a subset of those subtrees and get relatively accurate results. However, finding subtrees that share the same structure can become computationally demanding. Instead, we decided to just find leaf processes that were spawned using the same MFA and group them. So, instead of tracing all processes of each group, we just profile a randomly sampled subset. The sampling strategy that we used is very simple but still gives satisfactory results. When the processes of a group are more than a specified threshold (currently 50) we sample only a percentage (currently 10%) of them. The profiling results for these subsets are then scaled accordingly (i.e., multiplied by 10) to better represent the complete results.

It is important to note that the implementation of probabilistic tracing is based on the ERTS trace functionality, thus written in Erlang. We believe that the performance of our implementation could be significantly improved by implementing the process tree keeping and the process profiling in the ERTS.

3.3 Compiler+Loader

This is the component that is responsible for the compilation and loading of modules. It receives the collected profiling data (i.e., type information and function call data) from the controller, formats them, and calls an extended version of the HiPE compiler to compile the module and load it. The HiPE compiler is extended with two additional optimizations that are driven by the collected profiling data; these optimizations are described in Chapter 4.

There are several challenges that HiPErJiT has to deal with, in order to efficiently compile and load Erlang code. On the top of the list is *hot code loading*: the requirement to be able to replace modules, on an individual basis, while the system is running and without imposing a long stop to its operation. The second characteristic, which is closely related to hot code loading, is that the language makes a semantic distinction between module-local calls and *remote calls*, as also described in Section 2.1.2.

These two characteristics, combined with the fact that Erlang is primarily a concurrent language in which a large number of processes may be executing code from different modules at the same time, effectively impose that compilation happens in a module-at-a-time fashion, without opportunities for cross-module optimizations. The alternative, i.e., performing cross-module optimizations, implies that the runtime system must be able to quickly undo optimizations in the code of some module that rely on information from other modules, whenever those other modules change. Since such undoings can cascade arbitrarily deep, this alternative is not very attractive engineering-wise. In addition it requires that HiPErJiT automatically triggers the process of re-profiling for a module when it is reloaded.

The runtime system of Erlang/OTP supports hot code loading in a particular, arguably quite ad hoc, way. It allows for *up to two* versions of each module (m_{old} and $m_{current}$) to be simultaneously loaded, and redirects all remote calls to $m_{current}$, the most recent of the two. Whenever m_{new}, a new version of module m, is about to be loaded, all processes that still execute code of m_{old} are abruptly terminated, $m_{current}$ becomes the new m_{old}, and m_{new} becomes $m_{current}$. This quite elaborate mechanism is implemented by the code loader with support from ERTS, which has control over all Erlang processes.

It is important to note that the current code loading mechanism introduces a rather obscure but still possible race condition between HiPErJiT and the user. If the user tries to load a new version of a module after HiPErJiT has started compiling the current one, but before it has completed loading it, HiPErJiT could load JiT-compiled code for an older version of the module after the user has loaded the new one. Furthermore, if HiPErJiT tries to load a JiT-compiled version of a module m when there are processes executing m_{old} code, this could lead to processes being killed. Thus, HiPErJiT loads a module m as follows. First, it acquires a module lock, not allowing any other process to reload this specific module during that period. Then, it calls HiPE to compile the target module to native code. After compilation is complete, HiPErJiT repeatedly checks whether any process executes m_{old} code. When no process executes m_{old} code, the JiT-compiled version is loaded. Finally, HiPErJiT releases the lock so that new versions of module m can be loaded again. This way, HiPErJiT avoids that loading JiT-compiled code leads to processes being killed.

Of course, the above scheme does not offer any progress guarantees, as it is possible for a process to execute m_{old} code for an indefinite amount of time. In that case, the user would be unable to reload the module (e.g., after fixing a bug). In order to avoid this, HiPErJiT stops trying to load a JiT-compiled module after a user-defined timeout (the default value is 10 seconds), thus releasing the module lock.

Let us end this section with a brief note about decompilation. Most JiT compilers support decompilation, which is usually triggered when a piece of JiT-compiled code is not executed frequently enough anymore. The main benefit of doing this is that native code takes up more space than byte code, so decompilation often reduces the memory footprint of the system. However, since code size is not a big concern in today's machines, and since decompilation can increase the number of mode switches (which are known to cause performance overhead) HiPErJiT currently does not support decompilation.

Chapter 4

Runtime Data Driven Optimizations

A benefit of JiT compilers over AoT compilers is that they have access to run-time data, that are acquired during the execution of a program. This creates opportunities for more aggressive and beneficial optimizations, that cannot be made with compile-time data. Optimizations based on run-time data have been extensively studied, especially in the context of dynamic languages[Holz94, Bala00, Arno02, Arno05, Bolz09, Kedl13, Kedl14, Ren16]. In this chapter, we describe two optimizations that HiPErJiT performs, based on the collected profiling information, in addition to calling HiPE: type specialization and inlining.

4.1 Type Specializations

In dynamic languages, there is typically very little type information available during compilation. Types of values are determined at runtime and because of that, dynamic language compilers generate code that handles all possible value types by adding type tests to ensure that operations are performed on terms of correct type. Also all values are tagged, which means that their runtime representation contains information about their type. The combination of these two features leads to dynamic language compilers generating less efficient code than those for statically typed languages.

This problem has been tackled with static type analysis [Lind03, Sago03], runtime type feedback [Holz94], or a combination of both [Kedl13]. Runtime type feedback is essentially a mechanism that gathers type information from calls during runtime, and uses this information to create specialized versions of the functions for the most commonly used types. On the other hand, static type analysis tries to deduce type information from the code in a conservative way, in order to simplify it (e.g., eliminate some unnecessary type tests). In HiPErJiT, we employ a combination of these two methods.

4.1.1 Type Feedback

As also described above, type feedback consists of two main components:

- A component that collects run-time type information.
- A compiler pass that generates specialized code for the above type information.

Collecting Type Information

We will first describe how HiPErJiT collects the runtime type information. First of all, HiPErJiT contains a type tracer component, as also explained in Section 3.2, that gathers type information for function call arguments and return values during run-time. Consequently this is the main method of type information collection. However, as also described in Chapter 2, Erlang also allows users to decorate programs with type information for functions using type specs. As Erlang is a dynamically typed language these specs are not used by the compiler, but by tools that perform static analyses in order to detect discrepancies [Lind04]. Type specs should not be assumed correct by a compiler, because in case they are wrong the result could be erroneous or slow code. However in a JiT Compiler context these specs can be used as complement to the type information gathered during execution.

Considering all the above, we implemented a component that combines run-time type information with type specs. Type specs are mostly useful for functions which not enough runtime type information exist. A final note that has to be considered is that an inconsistency between the type spec and the dynamic type information means that the type spec is not correct. In that case HiPErJiT invalidates the spec and does not use it at all, but does not issue any warning to the user.

Optimistic Type Compilation

After profiling the argument types of function calls, the collected type information is used. However, since this information is approximate or may not hold for all subsequent calls, the optimistic type compilation pass adds type tests that check whether the arguments have the appropriate types. Its goal is to create specialized (optimistic) function versions, whose arguments are of known types.

Optimistic type compilation is the first compiler pass that is performed in Core Erlang, as it improves the benefit from many other optimizations. Its implementation is straightforward. For each function `f` where the collected type information is non-trivial:

- The function is duplicated into an optimized `f$opt` and a "standard" `f$std` version of the function.
- A header function that contains all the type tests is created. This function performs all necessary type tests for each argument, to ensure that they satisfy any assumptions upon which type specialization may be based. If all tests pass, the header calls `f$opt`, otherwise it calls `f$std`.
- The header function is inlined in every local function call of the specified function `f`. This ensures that the type tests happen on the caller's side, thus improving the benefit from the type analysis pass that happens later on.

4.1.2 Type Analysis

Optimistic type compilation on its own does not offer any performance benefit. It simply duplicates the code and forces execution of possibly redundant type tests. Its benefits arise from its combination with type analysis and propagation.

Type analysis is an optimization pass performed on Icode that infers type information for each program point and then simplifies the code based on this information. It removes checks and type tests that are unnecessary based on the inferred type information. It also removes some boxing and unboxing operations from floating-point computations. A brief description of the type analysis algorithm [Lind05] is as follows:

1. Construct the call graph for all the functions in a module and sort it topologically based on the dependencies between its strongly connected components (SCCs).
2. Analyze the SCCs in a bottom-up fashion using a constraint-based type inference to find the most general success typings [Lind06] under the current constraints.
3. Analyze the SCCs in a top-down order using a data-flow analysis to propagate type information from the call sites to module-local functions.

4. Add new constraints for the types, based on the propagated information from the previous step.

5. If a fix-point has not been reached, go back to step 2.

Initial constraints are mostly generated using the type information of Erlang Built-In Functions (BIFs) and functions from the standard library which are known to the analysis. Guards and pattern matches are also used to generate type constraints.

We present a simple example that showcases the algorithm and the importance of combining the bottom-up type inference with the top-down data flow analysis [Lind05].

```
-module(reverse).
-export([reverse/1]).

reverse(L) -> reverse(L, []).

reverse([H|T], Acc) -> reverse(T, [H|Acc]);
reverse([], Acc) -> Acc.
```

Listing 4.1: A module that contains the reverse list function

As the SCCs of the call graph are analyzed in a bottom-up fashion, the function `reverse/2` has to be analyzed first since it only depends on itself. Performing the type inference yields the following success typings (in the order with which the functions are analyzed):

```
reverse/2 :: (list(),any()) -> any()
reverse/1 :: (list()) -> any()
```

Listing 4.2: The success typings of reverse after type inference

Those signatures are correct but overestimate the type of the second argument of `reverse/2`. By applying the data-flow analysis for local functions, the type of the second argument of `reverse/2` is narrowed down to a `list()` as `reverse/2` is only called by `reverse/1` with a list as a second argument. The final success typing are shown below:

```
reverse/2 :: (list(),list()) -> list()
reverse/1 :: (list()) -> list()
```

Listing 4.3: The final success typings of reverse

This example is fairly simple and so a fix-point is reached after a single pass of the algorithm.

After type analysis, code optimizations are performed. First, all redundant type tests or other checks are completely removed. Then some instructions, such as floating-point arithmetic, are simplified based on the available type information. Finally, control flow is simplified and dead code is removed.

It is important to note that type analysis and propagation is restricted to the module boundary. No assumptions are made about the arguments of the exported functions, as those functions can be called from modules which are not yet present in the system or available for analysis. Thus, modules that export only few functions benefit more from this analysis as more constraints are generated for their local functions and used for type specializations.

An Example

We present a simple example that illustrates the benefit of type specialization. Listing 4.4 contains a function that computes the power of two values, where the base of the exponentiation is a number (an integer or a float) and the exponent is an integer.

```
-spec power(number(), integer(), number()) -> number().
power(_V1, 0, V3) -> V3;
power(V1, V2, V3) -> power(V1, V2-1, V1*V3).
```

Listing 4.4: The source code of a simple power function.

Without the optimistic type compilation, HiPE generates the Icode shown in Listing 4.5.

```
power/3(v1, v2, v3) ->
12:
    v5 := v3
    v4 := v2
    goto 1
1:
    _ := redtest()    (primop)
    if is_{integer,0}(v4) then goto 3 else goto 10
3:
    return(v5)
10:
    v8 := '-'(v4, 1)  (primop)
    v9 := '*'(v1, v5) (primop)
    v5 := v9
    v4 := v8
    goto 1
```

Listing 4.5: Generated Icode for the power function.

If we consider that this function is mostly called with a floating point number as the first argument, then HiPE with optimistic type compilation generates the Icode in Listing 4.6. Note that `power$std` has the same Icode as `power` without optimistic type compilation.

```
power/3(v1, v2, v3) ->
16:
    _ := redtest()    (primop)
    if is_float(v1) then goto 3 else goto 14
3:
    if is_integer(v2) then goto 5 else goto 14
5:
    if is_float(v3) then goto 7 else goto 14
7:
    power$opt/3(v1, v2, v3)
14:
    power$std/3(v1, v2, v3)

power$opt/3(v1, v2, v3) ->
24:
    v5 := v3
    v4 := v2
    goto 1
1:
    _ := redtest() (primop)
    if is_{integer,0}(v4) then goto 3 else goto 20
3:
    return(v5)
20:
    v8 := '-'(v4, 1)  (primop)
    _ := gc_test<3>() (primop) -> goto 28, #fail 28
28:
    fv10 := unsafe_untag_float(v5) (primop)
    fv11 := unsafe_untag_float(v1) (primop)
    _ := fclearerror()             (primop)
    fv12 := fp_mul(fv11, fv10)     (primop)
    _ := fcheckerror()             (primop)
    v5 := unsafe_tag_float(fv12)   (primop)
    v4 := v8
    goto 1
```

Listing 4.6: Generated Icode for the power function with optimistic type compilation.

As it can be seen, optimistic type compilation has used type propagation and found that both `v1` and `v3` are always floats. The code was then modified so that they are untagged unsafely in every iteration and multiplied using a floating point instruction, whereas without optimistic type compilation they are multiplied using the standard general multiplication function, which checks the types of both arguments and untags (and unboxes) them before performing the multiplication.

4.1.3 Discussion

Combining the optimistic type compilation, that HiPErJiT performs, with the type analysis pass in HiPE seems to reduce the execution times of most programs by a small amount. Its most important benefit, though, is that it allows for further optimizations to be performed on the optimistic branch of the code. Especially when combined with profile driven inlining, optimistic type compilation yields valuable performance improvements.

However there is still a lot of work to be done in order to improve optimistic type compilation, as we have only scratched the surface of its potential. Some possible lines of future work are briefly described below.

First of all, there is need to dynamically ensure that the optimistic CFG is indeed visited more frequently than the original CFG. Optimistic type compilation is based on type information gathered through profiling, so it is possible that some type assumptions that it makes are wrong. In case the optimistic CFG is found to be rarely (or never) executed, the function should be recompiled without the optimistic type compilation.

Another problem is that some typetests are executed multiple times in the non-optimistic CFG. An example where that problem is present follows below.

```
%% A type test added by our optimistic type compilation
foo/1(v1) ->
1:
  if is_nil(v1) then goto 2 else goto 3
2:
  foo$opt/1(v1)
3:
  foo$std/1(v1)

%% The standard function
foo$std/1(v1) ->
1:
  if is_nil(v1) then goto 2 else ...
2:
  return v1
...

%% The optimistic function
foo$opt/1(v1) ->
```

```
  return v1
```

Listing 4.7: An example of typetest redundancy.

In the above example the HiPE Type Analysis pass removed the redundant `is_nil` typetest in the optimistic `foo$opt` as `v1` has already passed an `is_nil` typetest. However the typetest in the standard `foo$std` would not be removed even though it is redundant. That is because Type Analysis does not deal with negation types. In the future, we would like to experiment with negation types, and potentially extend the Type Analysis pass to support them.

Finally we would like to experiment with more aggressive type optimizations on the optimistic CFGs.

4.2 Inlining

Inlining is the process of replacing a function call with the body of the called function. This improves performance in two ways. First, it mitigates the function call overhead. Second, and most important, it enables more optimizations to be performed later, as most optimizations usually do not cross function boundaries. That is the reason why inlining is usually performed in early phases of compilation so that later phases become more effective.

However, aggressive inlining has several potential drawbacks, both in compilation time, as well as in code size increase (which in turn can also lead to higher execution times because of caching effects). However, code size is less of a concern in today's machines, except of course in application domains where memory is not abundant (e.g., in IoT or embedded systems).

In order to get a good performance benefit from inlining, the compiler must achieve a fine balance between inlining every function call and not inlining anything at all. Therefore, the most important issue when inlining is choosing which function calls to inline. There has been a lot of work on how to make this decision with compile-time [Sant95, Serr97, Wadd97, Peyt02] as well as run-time information in the context of JiT compilers [Ayer97, Gran99, Suga02, Haze03]. HiPErJiT makes its inlining decisions based on run-time information, but also keeps its algorithm fairly lightweight so as to not to impose a big overhead.

4.2.1 Inlining Decision

We decided to use an inlining decision mechanism that uses runtime information to decide which function calls to inline. Our mechanism borrows two ideas from previous work, the use of call frequency [Ayer97] and call graphs [Suga02] to guide the inlining decision. Recall that HiPErJiT compiles whole modules at a time, thus inlining decisions are made based on information from all the functions in a module. Due to hot code loading, inlining across modules is not performed.

Finding the most profitable, performance-wise, function calls to inline and also the most efficient order in which to inline them is a heavy computational task and thus we decided to use heuristics to greedily decide which function calls to inline and when. The call frequency data that are used is the number of calls between each pair of function, as also described in Section 3.2.

Before describing our mechanism in detail we briefly describe the main idea and rationale behind its design.

Decisions are based on the assumption that call sites which are visited the most are the best candidates for inlining. Because of that our mechanism greedily chooses to inline the function pair (F_1, F_2) with most calls from F_1 to F_2.

Greedy inlining might introduce performance overheads as some calls might be inlined multiple times. For example, consider the case where our mechanism decides to inline the following pairs in that order $[(F_1, F_2), (F_2, F_3), (F_1, F_3)]$. In that case function F_3 would be inlined two times in the function F_2, once in the original F_2 and once in the inlined copy of F_2 in F_1. This could have been prevented if F_3 were first inlined in F_2 and then F_2 were inlined in F_1. However, the analysis needed to achieve this is not cost effective. Thus, the inlining decision is made greedily, despite occasionally inlining a function more than once.

Of course, inlining has to be restrained so that it does not happen for every function call in the program. We achieve this by restricting the maximum code size of each module. Small modules are allowed to grow up to double their size, while larger ones are only allowed to grow by 10%.

4.2.2 Implementation

Inline Decision Algorithm

Inlining decision making is made iteratively until there are no more calls to inline or until the module has become larger than a specified threshold.

The state that the iteration acts upon consists of the following data structures.

- A priority queue that contains pairs of functions (F_1, F_2) and the number of calls $N_{F_2}^{F_1}$ from F_1 to F_2 for each such pair. This priority queue supports three basic operations:
  - Find-Maximum: which returns the pair with the maximum number of calls.
  - Delete-Maximum: which returns and deletes the pair with the maximum number of calls.
  - Update: which updates a pair with a new number of calls.
- A set that contains all calls (F_1, F_2) that have been already inlined. This exists to prevent inlining loops from happening. However it also prevents self-recursive calls from being inlined, which could potentially improve performance. We decided to keep this simple mechanism, despite the potentially lost benefit on performance.
- A map of the total calls to each function which is initially constructed for each f by adding the number of calls for all pairs on the priority queue $T_f = \sum_{f_i} N_f^{f_i}$.
- A map of the size of each function, which is used to compute whether the module has become larger than the specified limit.

The main loop works as follows:

1. Delete-Maximum from the priority queue.
2. Check that all the following conditions are satisfied for the specific pair of functions (F_1, F_2).
   - Not being already inlined.
   - Having a number of calls higher than a specified threshold.
   - Not making the module grow more than the specified limit.

3. If one of those conditions fails, then the loop continues.

4. Otherwise:

   (a) All the local calls to F_2 in F_1 are inlined.

   (b) The set of already inlined pairs is updated with the pair (F_1, F_2).

   (c) The map of function sizes is updated for function F_1 based on its new size after the inline.

   (d) The priority queue is updated in the following way. We find all the pairs (F_2, F_x) in the priority queue and also all the total calls T_{F_2} for F_2. Then we update every pair (F_1, F_x) in the priority queue with the call number $N_{F_x}^{F_1} + N_{F_2}^{F_1} * N_{F_x}^{F_2} / T_{F_2}$. In practice this means that every call that was done from F_2 will now be done from F_1.

Inlining Algorithm

The inlining algorithm that we used is fairly standard, except for some details specific to Icode. In order to properly describe the algorithm we will first define a slightly simpler form of Erlang Icode, Icode-Mini, in Fig. 4.1 so that we can focus on the semantics of the optimization without having to deal with unnecessary details.

$c \in Const$	Constants
$x \in Var$	Variables
$l \in Label$	Labels
$f \in Function$	Functions
$t \in Type$	Types

$$
\begin{aligned}
v &::= c \mid x \\
i &::= \text{label } l \mid \text{if } v \text{ then } l_t \text{ else } l_f \mid \text{type } t\ v\ l_t\ l_f \mid \text{goto } l \mid \text{move } v_d\ v_s \mid \\
&\quad\ \text{phi } v_d\ (< l_1, v_1 >, ..., < l_n, v_n >) \mid \text{call } v_d\ f\ (v_1, ..., v_n) \mid \\
&\quad\ \text{enter } f\ (v_1, ..., v_n) \mid \text{return } v \mid \text{begin-try } l_1\ l_2 \mid \text{end-try} \mid \\
&\quad\ \text{fail } (v_1, ..., v_n)\ l
\end{aligned}
$$

Figure 4.1: Icode-mini definition

First of all we need a function that updates all variables and labels to unique variables and labels. Let *vmap* be a function that maps variables in the callee function to new unique variables in the caller function. Let *lmap* be a function that maps labels in the callee function to new unique labels in the caller function.

$$\begin{aligned}
U(\text{label } l.is) &= \text{label } lmap(l).U(is)\\
U(\text{if } v \text{ then } l_t \text{ else } l_f.is) &= \text{if } vmap(v) \text{ then } lmap(l_t) \text{ else } lmap(l_f).U(is)\\
U(\text{type } t\ v\ l_t\ l_f.is) &= \text{type } t\ vmap(v)\ lmap(l_t)\ lmap(l_f).U(is)\\
U(\text{goto } l.is) &= \text{label } lmap(l).U(is)\\
U(\text{move } v_d\ v_s.is) &= \text{move } vmap(v_d)\ vmap(v_s).U(is)\\
U(\text{phi } v_d\ (< l_1, v_1 >, ...).is) &= \text{phi } vmap(v_d)\ (< lmap(l_1), vmap(v_1) >, ...).U(is)\\
U(\text{call } v_d\ f\ (v_1, ..., v_n).is) &= \text{call } vmap(v_d)\ f\ (vmap(v_1), ..., vmap(v_n)).U(is)\\
U(\text{enter } f\ (v_1, ..., v_n).is) &= \text{enter } f\ (vmap(v_1), ..., vmap(v_n)).U(is)\\
U(\text{return } v.is) &= \text{return } vmap(v).U(is)\\
U(\text{begin-try } l_1\ l_2.is) &= \text{begin-try } lmap(l_1)\ lmap(l_2).U(is)\\
U(\text{fail } (v_1, ..., v_n)\ l.is) &= \text{fail } (vmap(v_1), ..., vmap(v_n))\ lmap(l).U(is)
\end{aligned}$$

We also need to define a function that transforms each enter to a call and a return. In practice enters have the same functionality as a call and a return.

$$E(\text{enter } f\ (v_1, v_2, ..., v_k).is) = \text{call } v_e\ f\ (v_1, v_2, ..., v_k).\text{return } v_e.E(is)$$

We also need to define a function that transforms all return instructions to a move and a goto as the returns in the callee should not return from the caller, but should rather just move their values to the destination variable.

$$R(\text{return } v.is, v_d, l_d) = \text{move } v_d\ v.\text{goto } l_d.is$$

Consider that we will inline function f_2 in function f_1. Every function has a name, arguments, and a body, which is as a sequence of instructions. Let $body(f_2) = i_1.i_2.is$, $args(f_2) = a_1, a_2, ..., a_m$, *vmap* and *lmap* as defined above. The final inlining transformation is the one below.

$$\begin{aligned}
I(\text{call } v_d\ f_2\ (v_1, v_2, ..., v_m).is) =& \text{move } vmap(a_1)\ v_1 ... \text{move } vmap(a_m)\ v_m.\\
& R(E(U(body(f_2))), v_d, l_d).\text{label } l_d.I(is)\\
I(\text{enter } f_2\ (v_1, v_2, ..., v_m).is) =& \text{move } vmap(a_1)\ v_1 ... \text{move } vmap(a_m)\ v_m\\
& U(body(f_2)).I(is)
\end{aligned}$$

The I transformation is only defined for the interesting cases, as in all other cases it functions as the identity function.

It is worth mentioning that Erlang uses cooperative scheduling which is implemented by making sure that processes are executed for a number of reductions and then yield. When the reductions of a process are depleted, the process is suspended and another process is scheduled in instead. Reductions are implemented in Icode by the `redtest()` primitive operation (primop) in the beginning of each function body; cf Listing 4.5. When inlining a function call, the call to `redtest()` in the body of the callee is also removed. This is safe to do, because inlining is bounded anyway.

Comparison with static inlining

In order to ensure that our profile driven inlining provides a real performance benefit, we compared it to the static inlining that the Erlang compiler offers. This static inlining algorithm is an implementation of the algorithm by Waddell and Dybvig [Wadd97], adapted to the Core Erlang language, with a slight change. Instead of always renaming variables and function variables, it uses the “no-shadowing” strategy [Peyt02]. We measured the execution times and the generated code sizes of several applications with static or profile driven inlining. The speedup of the different inlining methods compared to compilation without inlining is shown in Fig. 4.2. The code size growth of the different methods is shown in Fig. 4.3. The configurations that we compared are two static ones (a more conservative *Static-25* and a more aggressive one *Static-100*) and three profile driven ones (the number in those configurations is the minimum number of calls for a call to be inlined and *small/med* describe the allowed code size growth).

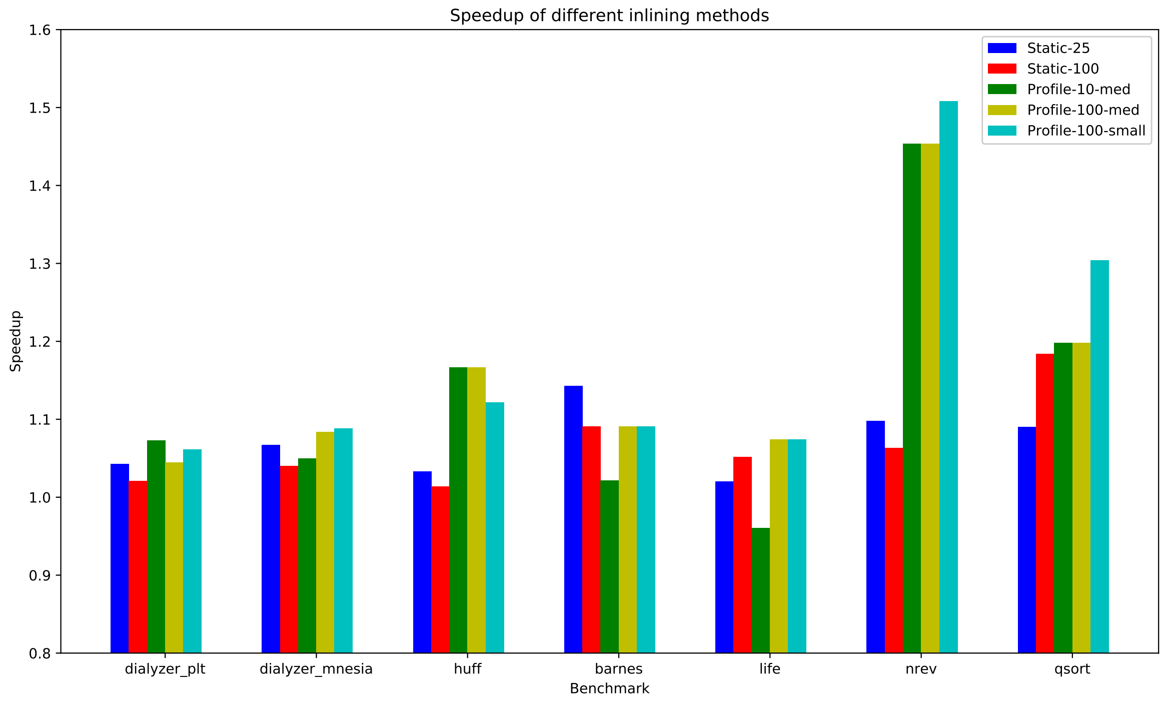

Figure 4.2: Speedup of different inlining methods

In Fig. 4.2 it can be seen that all different inlining methods provide a speedup when used. Note that higher speedup is better. Our profile driven inlining method generally leads to better execution times except for the barnes benchmark. We compared several different configurations for our inlining method and the more conservative ones, *Profile-100-small* and *Profile-100-med*, seem to perform better.

As can be seen in Fig. 4.3, two configurations of profile driven inlining lead to relatively large code growth. This can be mainly attributed to two causes. First of all, the inlining method *Profile-10-med* is very aggressive and inlines every call that has happened more than ten times (which leads to bigger code growth for the life benchmark). In addition, as also explained in Section 4.2.1, the maximum code growth limit is variable, so smaller modules can grow more than larger modules (especially in the *med* configurations). Therefore, applications huff, barnes, nrev, and qsort (which consist of small modules) have a larger limit and can grow much more.

It is interesting to note that *Static-25* generally does not lead to any code growth (in some cases it even reduces the code size). This can be attributed to two reasons: The performance from later optimizations is improved, thus reducing code size, and an analysis that removes non exported functions that are never called is performed after inlining. We have not yet implemented this analysis after our inlining pass so we expect to achieve better code size benefits after implementing it.

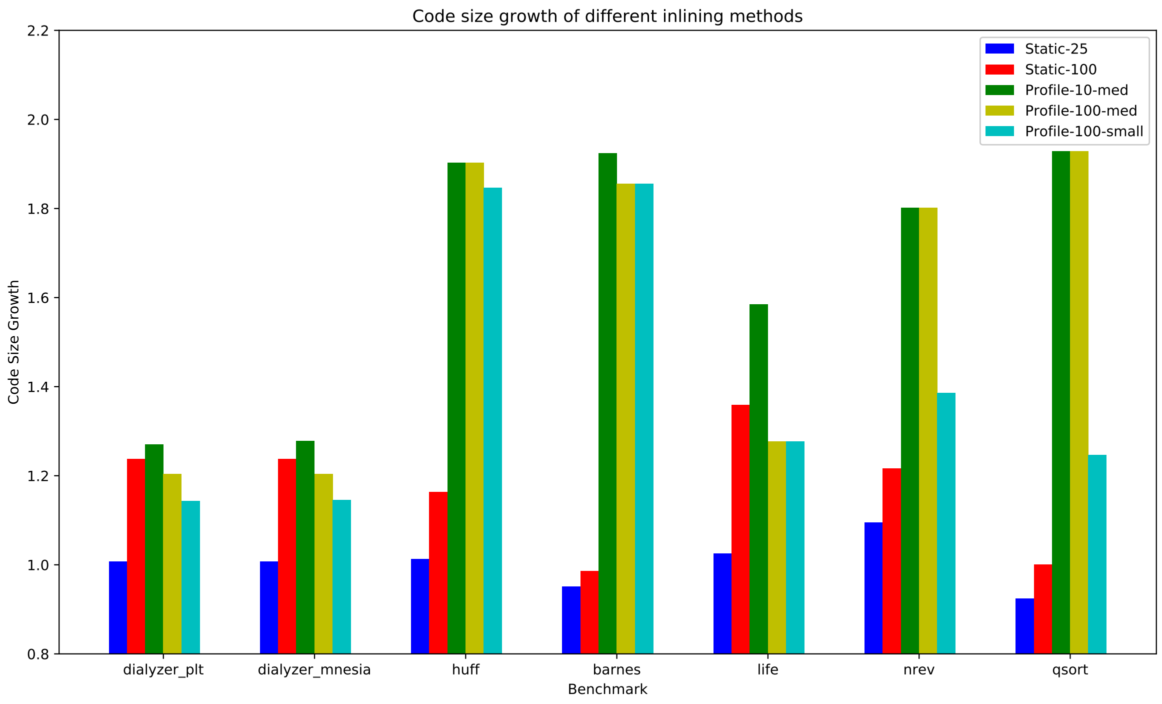

Figure 4.3: Code size growth of different inlining methods

Discussion and Further Work

Our inlining implementation generally achieves speedup compared to the static inlining, that is already implemented in the Erlang compiler, in both large and small applications. However there is still a lot of space for improvement. We do not yet inline tail-recursive calls, which could provide some additional benefit in some specific cases. In addition, we have not yet implemented a dead-function analysis after our inlining pass that checks whether a local function is called anywhere, and if not removes it.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of HiPErJiT against BEAM, which serves as a baseline for our comparison, and against two systems that also aim to surpass the performance of BEAM. The first of them is the HiPE compiler.[1] The second is the Pyrlang[2] meta-tracing JiT compiler for Erlang, which is a research prototype and not a complete implementation; we report its performance on the subset of benchmarks that it can handle. We were not able to obtain a version of BEAMJIT to include in our comparison, as this system is still (May 2018) not publicly available. However, as mentioned in Section 2.3.1, BEAMJIT does not achieve performance that is superior to HiPE anyway.

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

5.1 Profiling Overhead

Our first set of measurements concerns the profiling overhead that HiPErJiT imposes. To obtain a rough worst-case estimate, we used a modified version of HiPErJiT that profiles programs as they run but does not compile any module to native code. Note that this scenario is very pessimistic for HiPErJiT, because the common case is that JiT compilation will be triggered for some modules, HiPErJiT will stop profiling them at that point, and these modules will then most likely execute faster, as we will soon see. But even if native code compilation were not to trigger for any module, it would be very easy for HiPErJiT to stop profiling after a certain time period has passed or some other event (e.g., the number of calls that have been profiled has exceeded a threshold) has occurred. In any case, our measurements showed that the overhead caused by probabilistic profiling is around 10%. More specifically, the overhead we measured ranged from 5% (in most cases) up to 40% for some heavily concurrent programs.

We also separately measured the profiling overhead on all concurrent benchmarks with and without probabilistic profiling, to measure the benefit of probabilistic over standard profiling. The average overhead that standard profiling imposes on concurrent benchmarks is 19%. while the average slowdown of probabilistic profiling on such benchmarks is 13%.

5.1.1 Evaluation on Small Benchmark Programs

The benchmarks we used come from the ErLLVM benchmark suite[3], which has been previously used for the evaluation of HiPE [Joha00], ErLLVM [Sago12], BEAMJIT [Drej14], and Pyrlang [Huan16]. The benchmarks can be split in two sets: (1) a set of small, relatively simple but quite representative Erlang programs, and (2) the set of Erlang programs from

[1] For both BEAM and HiPE, we used the ‘master’ branch of Erlang/OTP 21.0.

[2] We used the latest version of Pyrlang (`https://bitbucket.org/hrc706/pyrlang/overview`) built using PyPy 5.0.1.

[3] https://github.com/cstavr/erllvm-bench

the Computer Language Benchmarks Game (CLBG)[4] as they were when the ErLLVM benchmark suite was created.

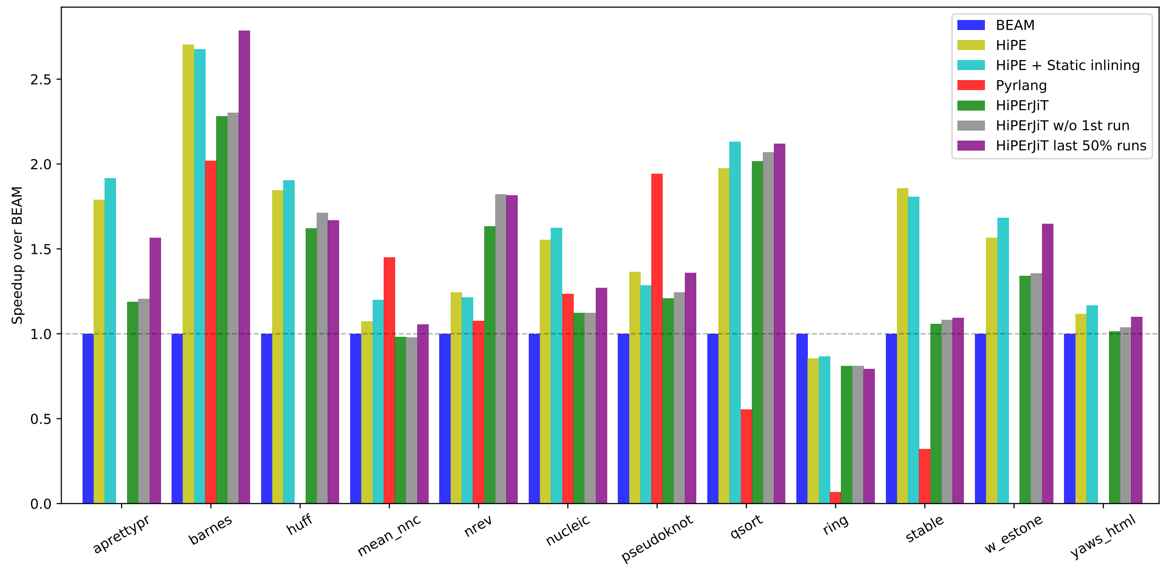

Figure 5.1: Speedup over BEAM on small benchmarks.

Comparing the performance of a Just-in-Time with an ahead-of-time compiler on small benchmarks is tricky, as the JiT compiler also pays the overhead of compilation during the program's execution. Moreover, a JiT compiler needs some time to warm up. For this reason, we report three numbers for HiPErJiT: the speedup achieved when considering all overheads, the speedup achieved when disregarding the first run, and the speedup achieved in the last 50% of many runs, when a steady state has been reached. We also use two configurations of HiPE: one with the maximum level of optimization (`o3`), and one where static inlining (using the `{inline_size,25}` compiler directive) has been performed besides `o3`.

The speedup of HiPErJiT, HiPE, and Pyrlang compared to BEAM for the small benchmarks is shown in Figs. 5.1 and 5.2. (We have split them into two figures based on scale of the y-axis, the speedup over BEAM.) Note that all speedups we report are averages of several different executions. The overall average speedup for each configuration is summarized in Table 5.1.

Table 5.1: 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

Overall, the performance of HiPErJiT is almost two times better than BEAM and Pyrlang, but slightly worse than HiPE. However, there also exist five benchmarks (`barnes`, `nrev`, `fib`, `smith` and `tak`) where HiPErJiT, in its steady state (the last 50% of runs), surpasses HiPE's performance. This strongly indicates that the profile-driven optimizations

[4] http://benchmarksgame.alioth.debian.org/

that HiPErJiT performs lead to more efficient code, compared to that of an ahead-of-time native code compiler performing static inlining.

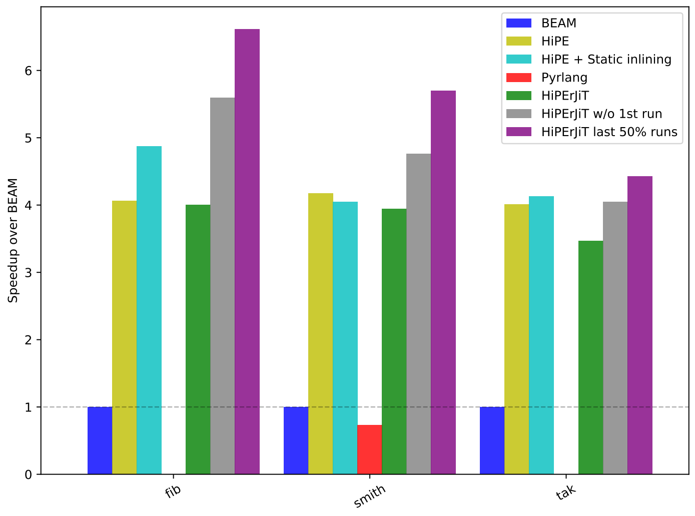

Figure 5.2: Speedup over BEAM on small benchmarks.

Out of those five benchmarks, the most interesting one is smith. It is an implementation of the Smith-Waterman DNA sequence matching algorithm. The reason why HiPErJiT offers better speedup over HiPE is that profile-driven inlining manages to inline functions `alpha_beta_penalty/2` and `max/2` in `match_entry/5`, which is the most time-critical function of the program, thus allowing further optimizations to improve performance. Static inlining on the other hand does not inline `max/2` in `match_entry/5` even if one chooses very large values for the inliner's thresholds.

On the ring benchmark (Fig. 5.2), which is heavily concurrent, HiPErJiT performs worse than both HiPE and BEAM. To be more precise, all systems perform worse than BEAM on this benchmark. The main reason is that the majority of the execution time is spent on message passing (around 1.5 million messages are sent per second), which is part of BEAM's runtime system. Finally, there are a lot of process spawns and exits (around 20 thousand per second), which leads to considerable profiling overhead, as HiPErJiT maintains and constantly updates the process tree.

Another interesting benchmark is stable (also Fig. 5.2), where HiPErJiT performs slightly better than BEAM but visibly worse than HiPE. This is mostly due to the fact that there are a lot of process spawns and exits in this benchmark (around 240 thousand per second), which leads to significant profiling overhead.

The speedup of HiPErJiT and HiPE compared to BEAM for the CLBG benchmarks is shown in Figs. 5.3 and 5.4. The overall average speedup for each configuration is shown in Table 5.2.

Table 5.2: Speedup over BEAM for the CLBG programs.

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

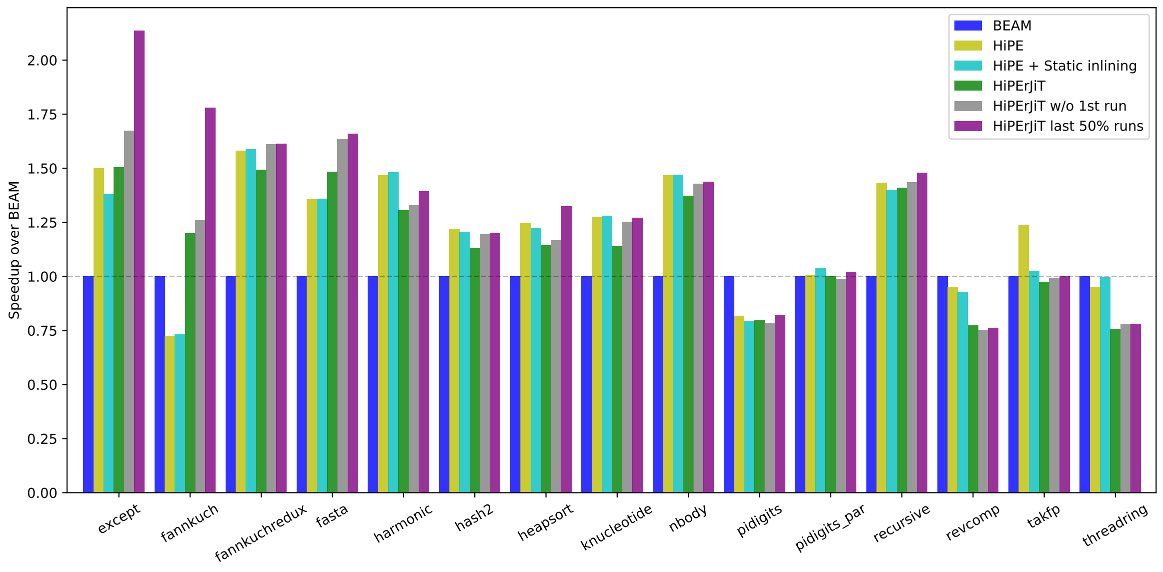

Figure 5.3: Speedup over BEAM on the CLBG programs.

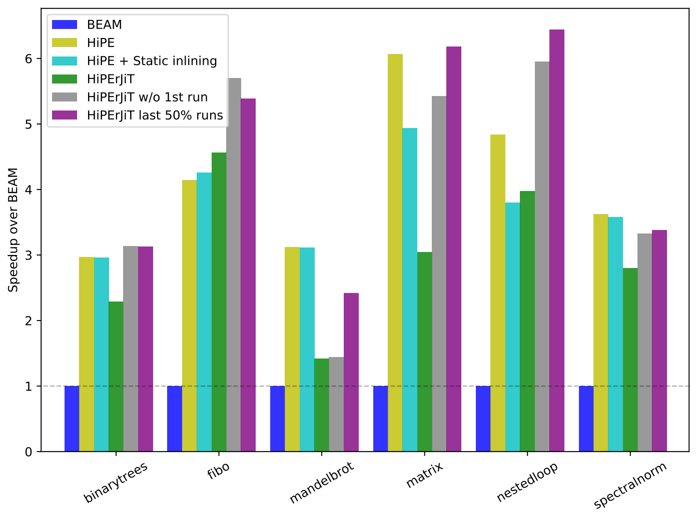

Figure 5.4: Speedup over BEAM on the CLBG programs.

As with the small benchmarks, the performance of HiPErJiT mostly lies between BEAM and HiPE. When excluding the first run, HiPErJiT outperforms both BEAM and HiPE in several benchmarks (binarytrees, except, fannkuch, fannkuchredux, fasta, fibo, nestedloop, and recursive). Finally, if we only consider the steady state of the JiT Compiler (last 50% of the runs), HiPErJiT outperforms both HiPE and BEAM in one more benchmark (matrix). However, HiPErJiT's performance on some benchmarks (revcomp, takfp, and threadring) is worse than both BEAM and HiPE because the profiling overhead is higher than the benefit of compilation.

5.1.2 Evaluation on a Bigger Program

Besides small benchmarks, we also evaluate the performance of HiPErJiT on a program of considerable size and complexity, as results in small or medium-sized benchmarks may not always provide a complete picture for the expected performance of a compiler. The Erlang program we chose, the Dialyzer [Lind04] static analysis tool, is both big (about 30, 000

LOC) and complex and highly concurrent [Aron13]. It has also been heavily engineered over the years and comes with hard-coded knowledge of the set of 26 modules it needs to compile to native code upon its start to get maximum performance for most use cases. Using an appropriate option, the user can disable this native code compilation phase, which takes more than a minute on the laptop we use, and in fact this is what we do to get measurements for BEAM and HiPErJiT.

We use Dialyzer in two ways. The first builds a Persistent Lookup Table (PLT) containing cached type information for all modules under erts, compiler, crypto, hipe, kernel, stdlib and syntax_tools. The second analyzes these applications for type errors and other discrepancies. The speedups of HiPErJiT and HiPE compared to BEAM for the two use ways of using Dialyzer are shown in Table 5.3.

Table 5.3: Speedups over BEAM for two Dialyzer use cases.

	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

The results show that HiPErJiT achieves performance which is better than BEAM's but worse than HiPE's. There are various reasons for this. First of all, Dialyzer is a complex application where functions from many different modules of Erlang/OTP (50–100) are called with arguments of significant size (e.g., the source code of the applications that are analyzed). This leads to considerable tracing and bookkeeping overhead. Second, some of the called modules contain very large, often compiler-generated, functions and their compilation takes considerable time (the total compilation time is about 70 seconds, which is a significant portion of the total time). Finally, HiPErJiT does not compile all modules from the start, which means that a percentage of the time is spent running interpreted code and performing mode switches which are more expensive than same-mode calls. In contrast, HiPE has hard-coded knowledge of "the best" set of modules to compile to native code before the analysis starts.

Chapter 6

Conclusion

6.1 The current state of HiPErJiT

We have presented HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on the HiPE native code compiler. It offers performance which is better than BEAM's and comparable to HiPE's, on most benchmarks. Aside from performance, we have been careful to preserve features such as hot code loading that are considered important for Erlang's application domain, and have made design decisions that try to maximize the chances that HiPErJiT remains easily maintainable and in-sync with components of the Erlang/OTP implementation. In particular, besides employing the HiPE native code compiler for most of its optimizations, HiPErJiT uses the same concurrency support that the Erlang Run-Time System provides, and relies upon the tracing infrastructure that it offers. Thus, it can straightforwardly profit from any improvements that may occur in these components.

Our main concern regarding HiPErJiT is the profiling overhead, especially in highly concurrent and complex applications. At the moment, it does not cause major problems because once HiPErJiT decides to compile a module to native code, the profiling stops and overhead drops to zero. However, in the context of continuous run-time optimization, which is the direction we want to pursue, profiling will become a critical issue.

6.2 Future Work

Despite the fact that the current implementation of HiPErJiT is quite robust and performs reasonably well, profile-driven JiT compilers are primarily engineering artifacts and can never be considered completely "done". Furthermore, HiPErJiT is just a first step towards the ultimate goal of lifelong feedback-directed optimization of programs. Therefore, there are several directions of work that we would like to follow in the future.

- 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.

Bibliography

[AdlT03] Ali-Reza Adl-Tabatabai, Jay Bharadway, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano and Tatiana Shpeisman, "The StarJIT compiler: A dynamic compiler for managed runtime environments", *Intel Technology Journal*, vol. 07, no. 01, 2003.

[Arms03] Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, Ph.D. thesis, Royal Institute of Technology Stockholm, Sweden, 2003.

[Arno00a] Ken Arnold, James Gosling, David Holmes and David Holmes, *The Java programming language*, vol. 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[Arno00b] Matthew Arnold, Stephen Fink, David Grove, Michael Hind and Peter F Sweeney, "Adaptive Optimization in the Jalapeño JVM", in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pp. 47–65, New York, NY, USA, 2000, ACM.

[Arno02] Matthew Arnold, Michael Hind and Barbara G Ryder, "Online Feedback-directed Optimization of Java", in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pp. 111–129, New York, NY, USA, 2002, ACM.

[Arno05] M Arnold, S J Fink, D Grove, M Hind and P F Sweeney, "A Survey of Adaptive Optimization in Virtual Machines", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, 2005.

[Aron13] Stavros Aronis and Konstantinos Sagonas, "On Using Erlang for Parallelization — Experience from Parallelizing Dialyzer", in *Trends in Functional Programming, 13th International Symposium, TFP 2012, Revised Selected Papers*, vol. 7829 of *LNCS*, pp. 295–310, Springer, 2013.

[Ayco03] John Aycock, "A Brief History of Just-in-time", *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[Ayer97] Andrew Ayers, Richard Schooler and Robert Gottlieb, "Aggressive Inlining", in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pp. 134–145, New York, NY, USA, 1997, ACM.

[Babc04] Brian Babcock, Mayur Datar and Rajeev Motwani, "Load Shedding for Aggregation Queries over Data Streams", in *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pp. 350–361, Washington, DC, USA, 2004, IEEE Computer Society.

[Bala00] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System", in *Proceedings of the ACM SIGPLAN*

2000 Conference on Programming Language Design and Implementation, PLDI ’00, pp. 1–12, New York, NY, USA, 2000, ACM.

[Baum15] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek and Sam Tobin-Hochstadt, “Pycket: A Tracing JIT for a Functional Language”, in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pp. 22–34, New York, NY, USA, 2015, ACM.

[Bebe10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann and Herman Venter, “SPUR: A Trace-based JIT Compiler for CIL”, in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, pp. 708–725, New York, NY, USA, 2010, ACM.

[Bolz09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo, “Tracing the Meta-level: PyPy’s Tracing JIT Compiler”, in *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS ’09, pp. 18–25, New York, NY, USA, 2009, ACM.

[Brig94] Preston Briggs, Keith D. Cooper and Linda Torczon, “Improvements to Graph Coloring Register Allocation”, *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, 1994.

[Burg97] Robert G Burger, *Efficient compilation and profile-driven dynamic recompilation in scheme*, Ph.D. thesis, Indiana University, 1997.

[Burg98] Robert G Burger and R Kent Dybvig, “An Infrastructure for Profile-Driven Dynamic Recompilation”, in *ICCL*, 1998.

[Carl04] Richard Carlsson, Thomas Lindgren, Björn Gustavsson, Sven-Olof Nyström, Robert Virding, Erik Johansson and Mikael Pettersson, “Core Erlang 1.0.3 language specification”, Technical report, Uppsala University, 2004.

[Cham89] Craig Chambers and David Ungar, “Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language”, in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI ’89, pp. 146–160, New York, NY, USA, 1989, ACM.

[Cham91] Craig Chambers and David Ungar, “Making Pure Object-oriented Languages Practical”, in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’91, pp. 1–15, New York, NY, USA, 1991, ACM.

[Cier00] Michał Cierniak, Guei-Yuan Lueh and James M Stichnoth, “Practicing JUDO: Java Under Dynamic Optimizations”, in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, pp. 13–26, New York, NY, USA, 2000, ACM.

[Deut84] Peter L. Deutsch and Allan M. Schiffman, “Efficient Implementation of the Smalltalk-80 System”, in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’84, pp. 297–302, New York, NY, USA, 1984, ACM.

[Drej14] Frej Drejhammar and Lars Rasmusson, "BEAMJIT: A Just-in-time Compiling Runtime for Erlang", in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pp. 61–72, New York, NY, USA, 2014, ACM.

[Eric] Ericsson AB, "EDoc - Erlang documentation generator".

[Fell15] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A McCarthy and Sam Tobin-Hochstadt, "The Racket Manifesto", in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 113–128, Dagstuhl, Germany, 2015, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Gal09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W Smith, Rick Reitmaier, Michael Bebenita, Mason Chang and Michael Franz, "Trace-based Just-in-time Type Specialization for Dynamic Languages", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 465–478, New York, NY, USA, 2009, ACM.

[Gran99] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers and Susan J Eggers, "An Evaluation of Staged Run-time Optimizations in DyC", in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pp. 293–304, New York, NY, USA, 1999, ACM.

[Gude93] David Gudeman, "Representing type information in dynamically typed languages", Technical report, University of Arizona, Department of Computer Science, 1993.

[Hans74] Gilbert Joseph Hansen, *Adaptive Systems for the Dynamic Run-time Optimization of Programs*, Ph.D. thesis, Carnegie-Mellon University, 1974.

[Harc97] Mor Harchol-Balter and Allen B Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing", *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[Haze03] Kim Hazelwood and David Grove, "Adaptive Online Context-sensitive Inlining", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pp. 253–264, Washington, DC, USA, 2003, IEEE Computer Society.

[Hejl06] Anders Hejlsberg, Scott Wiltamuth and Peter Golde, *C# Language Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Holz94] Urs Hölzle and David Ungar, "Optimizing Dynamically-dispatched Calls with Run-time Type Feedback", in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pp. 326–336, New York, NY, USA, 1994, ACM.

[Huan15] Ruochen Huang, Hidehiko Masuhara and Tomoyuki Aotani, "Pyrlang: A High Performance Erlang Virtual Machine Based on RPython", in *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pp. 48–49, New York, NY, USA, 2015, ACM.

[Huan16] Ruochen Huang, Hidehiko Masuhara and Tomoyuki Aotani, “Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler”, in *Post-Proceeding of the 17th Symposium on Trends in Functional Programming*, 2016.

[Hugh95] John Hughes, “The Design of a Pretty-printing Library”, in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 53–96, London, UK, 1995, Springer-Verlag.

[IEEE08] “IEEE Standard for Floating-Point Arithmetic”, 2008.

[Jime07] Miguel Jimenez, Tobias Lindahl and Konstantinos Sagonas, “A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings”, in *Proceedings of the 2007 SIGPLAN Workshop on Erlang*, Erlang ’07, pp. 11–17, New York, NY, USA, 2007, ACM.

[Joha96] Erik Johansson and Christer Jonsson, “Native Code Compilation for Erlang”, Technical report, Computing Science Department, Uppsala University, October 1996.

[Joha00] Erik Johansson, Mikael Pettersson and Konstantinos Sagonas, “A High Performance Erlang System”, in *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP ’00, pp. 32–43, New York, NY, USA, 2000, ACM.

[Joha03] Erik Johansson, Mikael Pettersson, Konstantinos Sagonas and Thomas Lindgren, “The development of the HiPE system: design and experience report”, *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 421–436, 2003.

[Kedl13] Madhukar N Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi and Ben Hardekopf, “Improved Type Specialization for Dynamic Scripting Languages”, in *Proceedings of the 9th Symposium on Dynamic Languages*, DLS ’13, pp. 37–48, New York, NY, USA, 2013, ACM.

[Kedl14] Madhukar N Kedlaya, Behnam Robatmili, C&#289;lin Ca\cscaval and Ben Hardekopf, “Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines”, in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’14, pp. 103–114, New York, NY, USA, 2014, ACM.

[Kenn07] Andrew Kennedy, “Compiling with Continuations, Continued”, in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’07, pp. 177–190, New York, NY, USA, 2007, Cambridge University Press.

[Kist03] Thomas Kistler and Michael Franz, “Continuous Program Optimization: A Case Study”, *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[Kotz08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell and David Cox, “Design of the Java HotSpot client compiler for Java 6”, *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 1, pp. 1–32, 2008.

[Kulk11] Prasad A Kulkarni, “JIT Compilation Policy for Modern Machines”, in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming*

Systems Languages and Applications, OOPSLA '11, pp. 773–788, New York, NY, USA, 2011, ACM.

[Latt04] Chris Lattner and Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pp. 75—-, Washington, DC, USA, 2004, IEEE Computer Society.

[Lind03] Tobias Lindahl and Konstantinos Sagonas, "Unboxed Compilation of Floating Point Arithmetic in a Dynamically Typed Language Environment", in *Proceedings of the 14th International Conference on Implementation of Functional Languages*, IFL'02, pp. 134–149, Berlin, Heidelberg, 2003, Springer Berlin Heidelberg.

[Lind04] Tobias Lindahl and Konstantinos Sagonas, "Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story", in Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, pp. 91–106, Berlin, Heidelberg, 2004, Springer-Verlag.

[Lind05] Tobias Lindahl and Konstantinos Sagonas, "TypEr: A Type Annotator of Erlang Code", in *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, Erlang '05, pp. 17–25, New York, NY, USA, 2005, ACM.

[Lind06] Tobias Lindahl and Konstantinos Sagonas, "Practical Type Inference Based on Success Typings", in *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pp. 167–178, New York, NY, USA, 2006, ACM.

[Luna04] Daniel Luna, Mikael Pettersson and Konstantinos Sagonas, "HiPE on AMD64", in *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang*, pp. 38–47, New York, NY, USA, September 2004, ACM.

[Pett02] Mikael Pettersson, Konstantinos Sagonas and Erik Johansson, "The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation", in *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Proceedings*, vol. 2441 of *LNCS*, pp. 228–244, Berlin, Heidelberg, September 2002, Springer.

[Peyt02] Simon Peyton Jones and Simon Marlow, "Secrets of the Glasgow Haskell Compiler Inliner", *J. Funct. Program.*, vol. 12, no. 5, pp. 393–434, 2002.

[Reis05] F Reiss and J M Hellerstein, "Data Triage: an adaptive architecture for load shedding in TelegraphCQ", in *21st International Conference on Data Engineering (ICDE'05)*, pp. 155–156, 2005.

[Ren16] Brianna M Ren and Jeffrey S Foster, "Just-in-time Static Type Checking for Dynamic Languages", in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 462–476, New York, NY, USA, 2016, ACM.

[Sago03] Konstantinos Sagonas, Mikael Pettersson, Richard Carlsson, Per Gustafsson and Tobias Lindahl, "All You Wanted to Know About the HiPE Compiler: (but Might Have Been Afraid to Ask)", in *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, Erlang '03, pp. 36–42, New York, NY, USA, 2003, ACM.

[Sago12] Konstantinos Sagonas, Chris Stavrakakis and Yiannis Tsiouris, “ErLLVM: an LLVM Backend for Erlang”, in *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pp. 21–32, New York, NY, USA, 2012, ACM.

[Sant95] Andre Santos, *Compilation by transformation for non-strict functional languages*, Ph.D. thesis, University of Glasgow, Scotland, 1995.

[Serr97] Manuel Serrano, “Inline expansion: When and how?”, in *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP ’97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, September 3–5, 1997 Proceedings*, pp. 143–157, Berlin, Heidelberg, 1997, Springer Berlin Heidelberg.

[Stee77] Guy Lewis Steele Jr., “Debunking the ”Expensive Procedure Call” Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO”, in *Proceedings of the 1977 Annual Conference*, ACM ’77, pp. 153–162, New York, NY, USA, 1977, ACM.

[Suga00] Toshio Suganuma, T Ogasawara, M Takeuchi, Toshiaki Yasue, M Kawahito, K Ishizaki, H Komatsu and Toshio Nakatani, “Overview of the IBM Java Just-in-time Compiler”, *IBM Syst. J.*, vol. 39, no. 1, pp. 175–193, 2000.

[Suga02] Toshio Suganuma, Toshiaki Yasue and Toshio Nakatani, “An Empirical Study of Method In-lining for a Java Just-in-Time Compiler”, in *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pp. 91–104, Berkeley, CA, USA, 2002, USENIX Association.

[Tama83] Hisao Tamaki and Taisuke Sato, “Program transformation through meta-shifting”, *New Generation Computing*, vol. 1, no. 1, pp. 93–98, 1983.

[Vird96] Robert Virding, Claes Wikström and Mike Williams, *Concurrent Programming in ERLANG (2nd Ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[Wadd97] Oscar Waddell and R Kent Dybvig, “Fast and effective procedure inlining”, in Pascal Van Hentenryck, editor, *Static Analysis*, pp. 35–52, Berlin, Heidelberg, 1997, Springer Berlin Heidelberg.