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

Konstantinos Kallas
University of Pennsylvania
Philadelphia, USA
kallas@seas.upenn.edu

Konstantinos Sagonas
Uppsala University
Uppsala, Sweden
kostis@it.uu.se

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.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Functional languages*; *Concurrent programming languages*;

KEYWORDS

Just-in-Time compilation, profile-driven optimization, HiPE, Erlang

ACM Reference Format:
Konstantinos Kallas and Konstantinos Sagonas. 2018. HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310234

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 that produces portable and reasonably efficient code for its virtual machine, called BEAM. 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.

This paper makes a first big step in that direction. We have developed HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on HiPE. 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.

The main part of this paper presents the architecture of HiPErJiT and the rationale behind some of our design decisions (Section 3), the profile-driven optimizations that HiPErJiT performs (Section 4), and the performance it achieves (Section 5). Before all that, in the next section, we review the current landscape of Erlang compilers. Related work is scattered throughout.

2 ERLANG AND ITS COMPILERS

In this section, we present a brief account of the various compilers for Erlang. Our main aim is to set the landscape for our work and present a comprehensive high-level overview of the various compilers that have been developed for the language, rather than a detailed technical one. For the latter, we refer the reader to the sites of these compilers and to publications about them.

2.1 JAM and BEAM

Technically, JAM and BEAM are abstract (virtual) machines for Erlang, not compilers. However, they both have lent their name to compilers that generate byte code for these machines. JAM is the older one, but since 1998, when the Erlang/OTP system became open source, the system has been exclusively based on BEAM.

The BEAM compiler is a fast, module-at-a-time compiler that produces relatively compact byte code, which is then loaded into the Erlang Run-Time System and expanded to indirectly threaded code for the VM interpreter (called "emulator" in the Erlang community). In the process, the BEAM loader specializes and/or merges byte

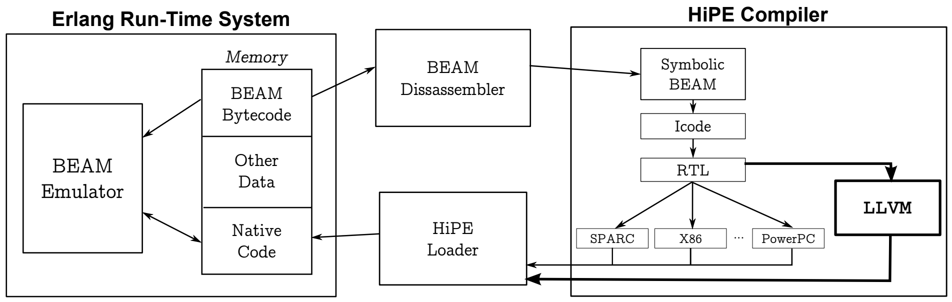

Figure 1: Architecture of some Erlang/OTP components: ERTS, HiPE and ErLLVM (from [24]).

code instructions. By now, the BEAM compiler comes with a well-engineered VM interpreter offering good performance for many Erlang applications. As a result, some other languages (e.g., Elixir) have chosen to use BEAM as their target. In recent years, the term "BEAM ecosystem" has been used to describe these languages, and also to signify that BEAM is important not only for Erlang.

2.2 HiPE and ErLLVM

The High-Performance Erlang (HiPE) compiler [13, 23] is an ahead-of-time native code compiler for Erlang. It has backends for SPARC, x86 [21], x86_64 [20], PowerPC, PowerPC 64, and ARM, has been part of the Erlang/OTP distribution since 2002, and is mature by now. HiPE aims to improve the performance of Erlang applications by allowing users to compile their time-critical modules to native code, and offer flexible integration between interpreted and native code. Since 2014, the ErLLVM compiler [24], which generates native code for Erlang using the LLVM compiler infrastructure [15] (versions 3.5 or higher), has also been integrated into HiPE as one of its backends, selectable by the `to_llvm` compiler option. In general, ErLLVM offers similar performance to the native HiPE backends [24].

Figure 1 shows how the HiPE compiler fits into Erlang/OTP. The compilation process typically starts from BEAM byte code. HiPE uses three intermediate representations, namely Symbolic BEAM, Icode, and RTL, and then generates target-specific assembly either directly or outsources its generation to ErLLVM.

Icode is a register-based intermediate language 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. The call stack is implicit and preserves registers. Finally, as Icode is a high-level intermediate language, bookkeeping operations (such as heap overflow checks and context switching) are implicit.

RTL is a generic three-address register transfer language. Call stack management and the saving and restoring of registers before and after calls are made explicit in RTL. Heap overflow tests are also made explicit and are propagated backwards in order to get merged. Registers in RTL are separated in tagged and untagged, where the untagged registers hold raw integers, floating-point numbers, and addresses.

RTL code is then translated to machine-specific assembly code (or to LLVM IR), virtual registers are assigned to real registers, symbolic references to atoms and functions are replaced by their real values in the running system, and finally, the native code is loaded into the Erlang Run-Time System (ERTS) by the HiPE loader.

ERTS allows seamless interaction between interpreted and native code. However, there are differences in the way interpreted and native code execution is managed. Therefore, a transition from native to interpreted code (or vice versa) triggers a *mode switch*. Mode switches occur in function calls, returns, and exception throws between natively-compiled and interpreted functions. HiPE was designed with the goal of no runtime overhead as long as the execution mode stays the same, so mode switches are handled by instrumenting calls with special mode switch instructions and by adding extra call frames that cause a mode switch after each function return. Frequent mode switching can negatively affect execution performance. Therefore, it is recommended that the most frequently executed functions of an application are all executed in the same mode [23].

2.3 BEAMJIT and Pyrlang

Unsurprisingly, JiT compilation has been investigated in the context of Erlang several times in the past, both distant and more recent. For example, both Jerico [12], 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. The first of them, BEAMJIT [6], 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 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 [6]. 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" [6, 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.[1] Since BEAMJIT is not available, not even as a binary, we cannot compare against it.

[1]Lukas Larsson (member of Erlang/OTP team), private communication, May 2018.

The second attempt, Pyrlang [11], is an alternative virtual machine for the BEAM byte code which uses RPython's meta-tracing JiT compiler [5] 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 Pyrlang paper [11] 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, not 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 Section 5.

2.4 Challenges of Compiling Erlang

The Erlang programming language comes with some special characteristics which make its efficient compilation quite challenging. 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, 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.

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.

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.

Unlike BEAMJIT and Pyrlang, we decided, at least for the time being, to leave the Erlang Run-Time System unchanged as far as hot code loading is concerned. This also means that, like HiPE, the unit of JiT compilation of HiPErJiT is the entire module.

3 HIPERJIT

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, organizes it, and propagates it 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. 2.

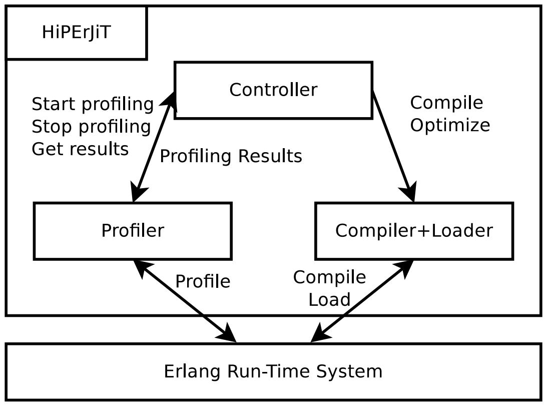

Figure 2: High-level architecture of HiPErJiT.

3.1 Controller

The controller, as stated above, is the fundamental component 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 Jalapeño JVM [1].

Traditionally, many JiT compilers use a lightweight call frequency method to drive compilation [3]. 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 accurately 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 [8] show that the total execution time of UNIX processes

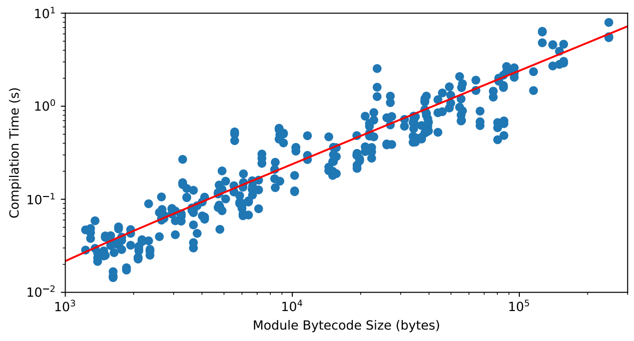

Figure 3: Estimating compilation time as a function of module byte code size. Observe that both axes are log-scale.

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. Motivated by those results, and 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 $CompTime = C * Size$. Based on the above assumptions, 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 all the benchmark programs, that were used for evaluation, 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 all the modules of the above programs, measured their compilation time, and fitted a line to the set of given points using the Least Squares method (cf. Fig. 3).

The line has a slope of $2.5e^{-5}$ so that is also the estimated compilation cost (in seconds) 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 first 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 (one that has the same name and MD5 hash) is considered for compilation at a subsequent time, HiPErJiT will use the stored compilation time measurement in place of $CompTime$.

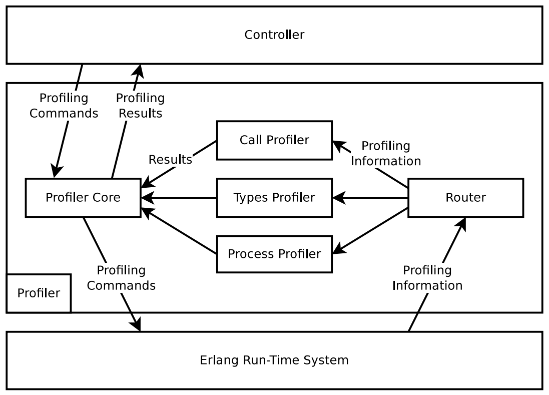

Figure 4: Profiler architecture and its components.

3.2 Profiler

The profiler is responsible for efficiently profiling executed code using the ERTS profiling infrastructure. Its architecture, which can be seen in Fig. 4, 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.

3.2.1 Profiling Execution Time. In order to profile the execution time that is being spent in each function, the profiler receives a time-stamped message from ERTS for each function call, function return, and process scheduling action. 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 $[(T_2 - T_1, F_1), (T_3 - T_2, F_2), \ldots, (T_n - T_{n-1}, F_{n-1})]$. The sum of all the differences for each function gives the total execution time spent in each function. An advantage of this method is that it allows us to also track the number of calls between each pair of functions, which is later used for profile-driven call inlining (Section 4.2).

The execution time profiler is able to handle the stream of messages sent during the execution of sequential programs. 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.

In order to tackle this problem, we implemented a probing mechanism that checks whether the number of unhandled messages in the mailbox of the execution time profiler exceeds a threshold (currently 1 million messages), in essence checking whether the arrival rate of messages is higher than their consumption rate. If the mailbox exceeds this size, no more trace messages are sent from ERTS until the profiler handles the remaining messages in the mailbox.

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.

The function call arguments to functions in modules that have been selected for profiling are recorded by HiPErJiT. 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 [18], 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 (lists and maps) 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 by considering only a small subset of their elements.

What is left is a way to generalize and aggregate the type information acquired through the profiling of many function calls. Types in Erlang are internally represented using a subtyping system [18], thus forming a type lattice. Because of that, a supremum operation 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 *genealogy based statistical profiling*, or genealogy profiling in short. 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 profiling all processes of each group, we just profile a randomly sampled subset. The sampling strategy that we used as a starting point 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.

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 Section 4.

There are several challenges that HiPErJiT has to deal with, when loading an Erlang module, as also mentioned in Section 2.4. First of all, Erlang supports hot code loading, so a user can reload the code of a module while the system is running. If this happens, HiPErJiT automatically triggers the process of re-profiling this module.

Currently ERTS allows only up to two versions of the same module to be simultaneously loaded. This 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.

4 PROFILE-DRIVEN OPTIMIZATIONS

In this section, 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 Specialization

In dynamically typed languages, there is typically very little type information available during compilation. Types of values are determined at runtime and because of that, compilers for these languages 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 compilers that generate less efficient code than those for statically typed languages.

This problem has been tackled with static type analysis [16, 23], runtime type feedback [10], or a combination of both [14]. 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 Optimistic Type Compilation. After profiling the argument types of function calls as described in Section 3.2.2, 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.

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 by HiPE 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 [18] 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 [19] 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.

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.

4.1.3 An Example. We present a simple example that illustrates type specialization. Listing 1 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 1: The source code of a simple power function.

Without the optimistic type compilation, HiPE generates the Icode shown in Listing 2.

```
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 2: 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 3. The Icode for `power$std` is not included as it is the same as the Icode for `power` without 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.

```
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 3: Generated Icode for the power function with optimistic type compilation.

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 [22, 25, 26, 28] as well as run-time information in the context of JiT compilers [4, 7, 9, 27]. 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. Our mechanism borrows two ideas from previous work, the use of call frequency [4] and call graphs [27] 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.

The main idea behind our decision mechanism is 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 F_b into F_a when:

$$\forall i, j. NumberOfCalls_{a,b} \geq NumberOfCalls_{i,j}$$

where $NumberOfCalls_{i,j}$ is the number of calls from F_i to F_j.

Of course, inlining has to be restrained, so that it does not happen for every function call in the program. We achieve this by limiting the maximum code size of each module as seen below:

$$MaxCodeSize = min\left(\frac{SmallCodeSize}{InitialCodesize} + 1.1, 2.1\right) * InitialCodeSize$$

We measured code size as the number of instructions in the Icode of a module and that $SmallCodeSize$ is the size of the smallest module in the benchmarks that were used for evaluation. To better clarify, a module m that has $InitialCodeSize_m = 2 * SmallCodeSize$, will be allowed to grow up to $MaxCodeSize_m = 1.6 * InitialCodeSize_m$. Note that the exact configuration of the maximum code size formula have been chosen after evaluating its performance against some alternatives. However, we have not extensively tuned it, and it could certainly be improved.

A detailed description of the decision algorithm follows. Inlining decisions are made iteratively until there are no more calls to inline or until the module has reached the maximum code size. The iteration acts on the following data structures.

- A priority queue that contains pairs of functions (F_i, F_j) and the number of calls $NoC_{i,j}$ from F_i to F_j for each such pair. This priority queue supports two basic operations:
  - Delete-Maximum: which deletes and returns the pair with the maximum number of calls.
  - Update: which updates a pair with a new number of calls.
- A map of the total calls to each function, which is initially constructed for each F_x by adding the number of calls for all pairs on the priority queue $TC_x = \sum_i NoC_{i,x}$.
- A map of the code 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 pair of functions (F_x, F_y) from the priority queue.
2. Check whether inlining the pair (F_x, F_y) makes the module exceed the maximum code size limit. If it does, then the loop continues, otherwise:
   (a) All the local calls to F_y in F_x are inlined.
   (b) The set of already inlined pairs is updated with the pair (F_x, F_y).
   (c) The map of function sizes is updated for function F_x based on its new size after the inline.
   (d) The number of calls of every pair (F_x, F_i) in the priority queue is updated to $NoC_{x,i} + NoC_{x,y} * NoC_{y,i}/TC_y$. In practice this means that every call that was done from F_y will now be also done from F_x.

4.2.2 Implementation. Our inlining implementation is fairly standard except for some details specific to Icode. The steps below describe the inlining process. In case the inlined call is a tail-call, only the first two steps need to be performed.

1. Variables and labels in the body of the callee are updated to fresh ones so that there is no name clash with the variables and labels in the body of the caller.
2. The arguments of the function call are moved to the parameters of the callee.
3. Tail-calls in the body of the callee are transformed to a normal call and a return.
4. Return instructions in the body of the callee are transformed into a move and a goto instruction that moves the return value to the destination of the call and jumps to the end of the body of the callee.

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 2. 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.

5 EVALUATION

In this section, 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.[2] The second is the Pyrlang[3] 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, 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

[2]For both BEAM and HiPE, we used the 'master' branch of Erlang/OTP 21.0.
[3]We used the latest version of Pyrlang (https://bitbucket.org/hrc706/pyrlang/overview) built using PyPy 5.0.1.

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, we executed all the benchmark programs, that were used for evaluation, using the modified version of HiPErJiT and our measurements showed that the overhead caused by genealogy 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 genealogy profiling, to measure the benefit of genealogy over standard profiling. The average overhead that standard profiling imposes on concurrent benchmarks is 19%, while the average overhead of genealogy profiling on such benchmarks is 13%.

5.2 Evaluation on Small Benchmark Programs

The benchmarks we used come from the ErLLVM benchmark suite[4], which has been previously used for the evaluation of HiPE [13], ErLLVM [24], BEAMJIT [6], and Pyrlang [11]. 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 the Computer Language Benchmarks Game (CLBG)[5] as they were when the ErLLVM benchmark suite was created.

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. Because of that, we have executed each benchmark enough times to allow HiPErJiT to compile the relevant application modules to native code. More specifically we run each benchmark $2 * N$ times, where after approximately N times HiPErJiT had compiled all relevant modules to native code. We report three numbers for HiPErJiT: the speedup achieved when considering the total execution time, the speedup achieved when disregarding the first run, and the speedup achieved in the last N runs, when a steady state has been reached.[6] 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 and 6. (We have split them into two figures based on scale of the y-axis, the speedup over BEAM.) Note that the speedups that we report are averages of five different executions of each benchmark multi-run. The black vertical lines indicate the standard deviation. The overall average speedup for each configuration is summarized in Table 1.

[4]https://github.com/cstavr/erllvm-bench
[5]http://benchmarksgame.alioth.debian.org/
[6]At the moment, HiPErJiT does not perform any code decompilation or deoptimization and therefore it always reaches a steady state after it has compiled and optimized all relevant modules.

Table 1: Speedup over BEAM for the small benchmarks.

Configuration	Speedup
HiPE	2.04
HiPE + Static inlining	2.09
Pyrlang	1.14
HiPErJiT	1.63
HiPErJiT w/o 1st run	1.98
HiPErJiT last 50% runs	2.31

Table 2: Speedup over BEAM for the CLBG programs.

Configuration	Speedup
HiPE	2.10
HiPE + Static inlining	2.10
HiPErJiT	1.77
HiPErJiT w/o 1st run	2.03
HiPErJiT last 50% runs	2.15

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

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. 6), where a message is cycled through a ring of processes, 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 handled by 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. 6), 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. 7 and 8. The overall average speedup for each configuration is shown in Table 2.

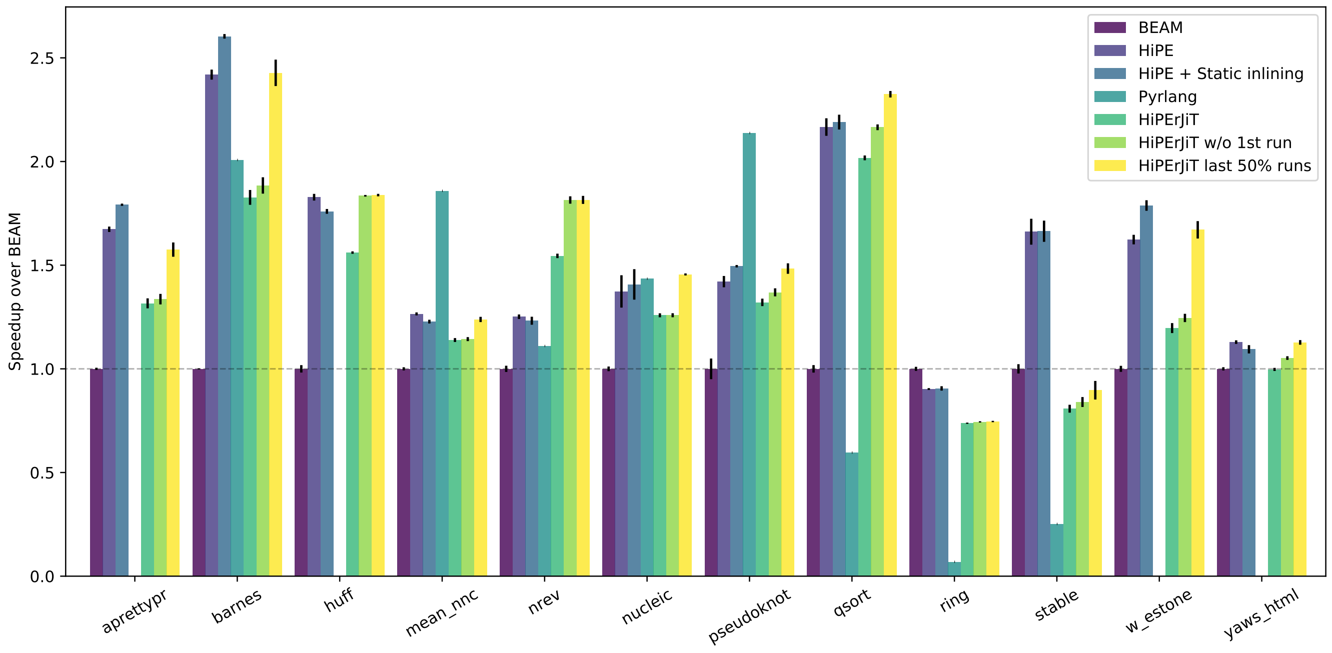

Figure 5: Speedup over BEAM on small benchmarks.

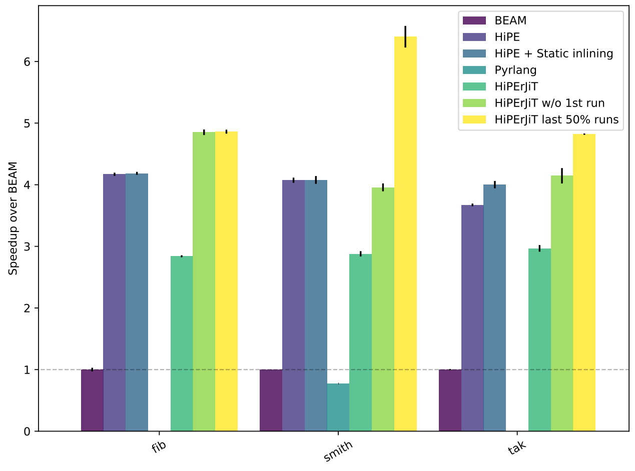

Figure 6: Speedup over BEAM on small benchmarks.

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 (except, fannkuchredux, fasta, fibo, and nestedloop). However, HiPErJiT's performance on some benchmarks (revcomp, and threadring) is worse than both BEAM and HiPE because the profiling overhead is higher than the benefit of compilation.

5.3 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 [17] static analysis tool, is big (about 30, 000 LOC), complex, and highly concurrent [2]. 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 3.

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).

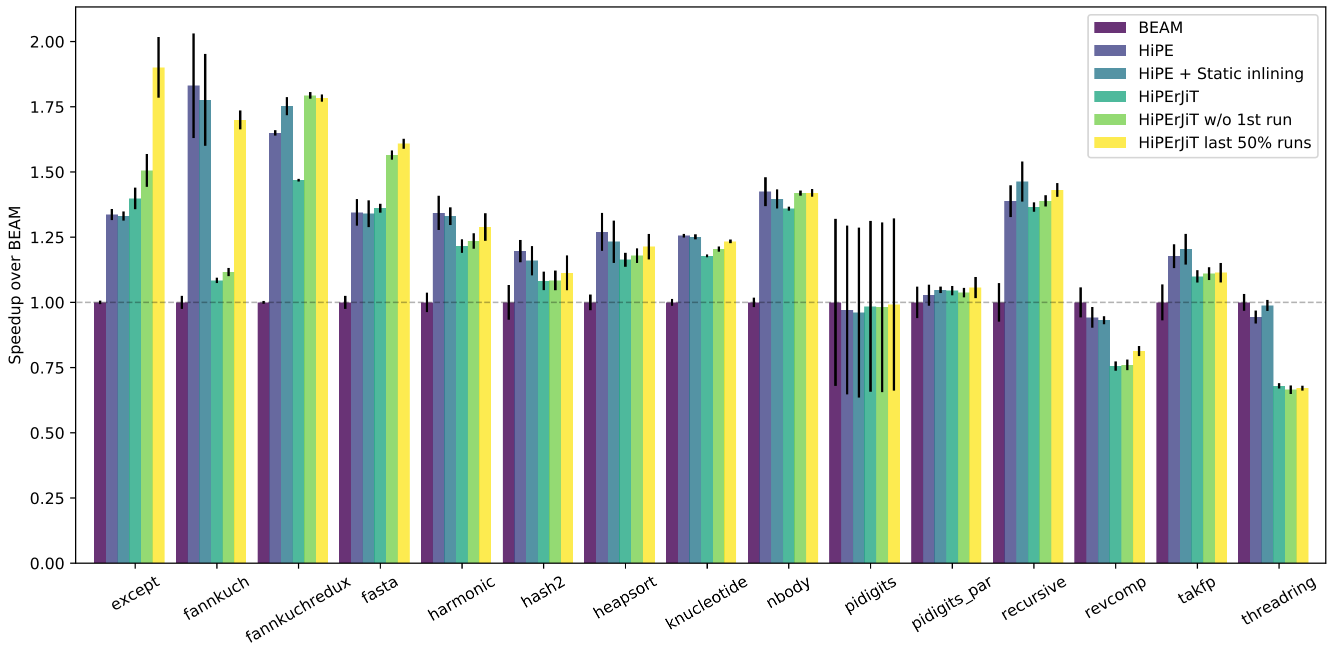

Figure 7: Speedup over BEAM on the CLBG programs.

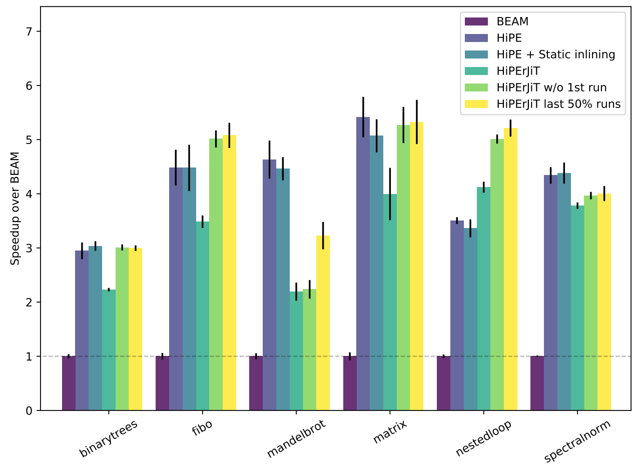

Figure 8: Speedup over BEAM on the CLBG programs.

Table 3: Speedups over BEAM for two Dialyzer use cases.

	Dialyzer Speedup	
Configuration	Building PLT	Analyzing
HiPE	1.78	1.73
HiPE + Static inlining	1.78	1.77
HiPErJiT	1.46	1.42
HiPErJiT w/o 1st run	1.61	1.55
HiPErJiT last 50% runs	1.67	1.60

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 natively compile before the analysis starts, and runs native code from the beginning of the analysis.

6 CONCLUDING REMARKS AND FUTURE WORK

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.

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”. Besides making the current optimizations more effective and implementing additional ones, one item which is quite high on our “to do” list is investigating techniques that reduce the profiling overhead, especially in heavily concurrent applications. For the current state of HiPErJiT, the profiling overhead is bigger than we would like it to be but, on the other hand, it's not really a major concern because once HiPErJiT decides to compile a module to native code the profiling of its functions stops and the overhead drops to zero. But it will become an issue if HiPErJiT becomes a JiT compiler that performs lifelong feedback-directed optimization of programs, which is a direction that we want to pursue.

ACKNOWLEDGMENTS

Most of this work was done at the National Technical University of Athens, Greece, as part of the first author's diploma thesis. The research time of the second author was partly supported by the Swedish Research Council (Vetenskapsrådet). We thank the IFL'2018 reviewers for comments that have improved the presentation of our work.

REFERENCES

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. 2000. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] Stavros Aronis and Konstantinos Sagonas. 2013. On Using Erlang for Parallelization — Experience from Parallelizing Dialyzer. In *Trends in Functional Programming, 13th International Symposium, TFP 2012, Revised Selected Papers (LNCS)*, Vol. 7829. Springer, 295–310. https://doi.org/10.1007/978-3-642-40447-4_19

[3] John Aycock. 2003. A Brief History of Just-in-time. *ACM Comput. Surv.* 35, 2 (2003), 97–113. https://doi.org/10.1145/857076.857077

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. 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)*. ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[6] Frej Drejhammar and Lars Rasmusson. 2014. BEAMJIT: A Just-in-time Compiling Runtime for Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang '14)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2633448.2633450

[7] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J Eggers. 1999. 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)*. ACM, New York, NY, USA, 293–304. https://doi.org/10.1145/301618.301683

[8] Mor Harchol-Balter and Allen B Downey. 1997. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Trans. Comput. Syst.* 15, 3 (1997), 253–285. https://doi.org/10.1145/263326.263344

[9] Kim Hazelwood and David Grove. 2003. Adaptive Online Context-sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 253–264. http://dl.acm.org/citation.cfm?id=776261.776289

[10] Urs Hölzle and David Ungar. 1994. 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)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[11] Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani. 2016. 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.* https://tfp2016.org/papers/TFP{_}2016{_}paper{_}16.pdf

[12] Erik Johansson and Christer Jonsson. 1996. *Native Code Compilation for Erlang.* Technical Report. Computing Science Department, Uppsala University.

[13] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. 2000. A High Performance Erlang System. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '00)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/351268.351273

[14] Madhukar N Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved Type Specialization for Dynamic Scripting Languages. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2508168.2508177

[15] Chris Lattner and Vikram Adve. 2004. 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)*. IEEE Computer Society, Washington, DC, USA, 75—-. http://dl.acm.org/citation.cfm?id=977395.977673

[16] Tobias Lindahl and Konstantinos Sagonas. 2003. 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)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–149. http://dl.acm.org/citation.cfm?id=1756972.1756981

[17] Tobias Lindahl and Konstantinos Sagonas. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, Wei-Ngan Chin (Ed.). Springer-Verlag, Berlin, Heidelberg, 91–106. https://doi.org/10.1007/978-3-540-30477-7_7

[18] Tobias Lindahl and Konstantinos Sagonas. 2005. TypEr: A Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang (Erlang '05)*. ACM, New York, NY, USA, 17–25. https://doi.org/10.1145/1088361.1088366

[19] Tobias Lindahl and Konstantinos Sagonas. 2006. 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)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1140335.1140356

[20] Daniel Luna, Mikael Pettersson, and Konstantinos Sagonas. 2004. HiPE on AMD64. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 38–47. https://doi.org/10.1145/1022471.1022478

[21] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Proceedings (LNCS)*, Vol. 2441. Springer, Berlin, Heidelberg, 228–244. https://doi.org/10.1007/3-540-45788-7_14

[22] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (2002), 393–434. https://doi.org/10.1017/S0956796802004331

[23] Konstantinos Sagonas, Mikael Pettersson, Richard Carlsson, Per Gustafsson, and Tobias Lindahl. 2003. 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)*. ACM, New York, NY, USA, 36–42. https://doi.org/10.1145/940880.940886

[24] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: an LLVM Backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*. ACM, New York, NY, USA, 21–32. https://doi.org/10.1145/2364489.2364494

[25] Andre Santos. 1995. *Compilation by transformation for non-strict functional languages*. Ph.D. Dissertation. University of Glasgow, Scotland. https://www.microsoft.com/en-us/research/publication/compilation-transformation-non-strict-functional-languages/

[26] Manuel Serrano. 1997. 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*. Springer Berlin Heidelberg, Berlin, Heidelberg, 143–157. https://doi.org/10.1007/BFb0033842

[27] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2002. 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*. USENIX Association, Berkeley, CA, USA, 91–104. http://dl.acm.org/citation.cfm?id=648042.744889

[28] Oscar Waddell and R Kent Dybvig. 1997. Fast and effective procedure inlining. In *Static Analysis*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–52.