Pallene: A statically typed companion language for Lua

Hugo Musso Gualandi
PUC-Rio
hgualandi@inf.puc-rio.br

Roberto Ierusalimschy
PUC-Rio
roberto@inf.puc-rio.br

ABSTRACT

The simplicity and flexibility of dynamic languages make them popular for prototyping and scripting, but the lack of compile-time type information makes it very challenging to generate efficient executable code.

Inspired by ideas from scripting, just-in-time compilers, and optional type systems, we are developing Pallene, a statically typed companion language to the Lua scripting language. Pallene is designed to be amenable to standard ahead-of-time compilation techniques, to inter-operate seamlessly with Lua (even sharing its runtime), and to be familiar to Lua programmers.

In this paper, we compare the performance of the Pallene compiler against LuaJIT, a just in time compiler for Lua, and with C extension modules. The results suggest that Pallene can compete with them in terms of performance.

1 INTRODUCTION

The simplicity and flexibility of dynamic languages make them popular for prototyping and scripting, but the lack of compile-time type information makes it very challenging to generate efficient executable code. There are at least three approaches to improve the performance of dynamically typed languages: scripting, just-in-time compilation, and optional type systems.

The scripting approach [19] advocates the use of two separate languages to write a program: a low-level *system language* for the parts of the program that need good performance and that interact with the underlying operating system, and a high-level *scripting language* for the parts that need flexibility and ease of use. The main advantage of the scripting approach is that the programmer can choose the programming language most suited for each particular task. The main disadvantages are due to the large differences between the languages. That makes it hard to rewrite a piece of code from one language to the other and also adds run-time overhead in the API between the two language runtimes.

Just-in-time compilers [6] dynamically translate high level code into low-level machine code during the program execution, on demand. To maximize performance, JIT compilers can collect runtime information, such as function parameter types, and use that information to generate efficient specialized code. The most appealing aspect of JIT compilers for dynamic languages is that they can provide a large speedup without the need to rewrite the code in a different language. In practice, however, this speedup is not always guaranteed and programmers often need to rewrite their code anyway, using idioms and "incantations" that are more amenable to optimization [10].

As the name suggests, optional type systems allow the programmer to partially add types to programs.[1] These systems combine the static typing and dynamic typing disciplines in a single language. From static types they seek to obtain better compile-time error checking, machine-checked lightweight documentation, and run-time performance; from dynamic typing they seek flexibility and ease of use. One of the selling points of optional type systems is that they promise a smooth transition from small dynamically typed scripts to larger statically typed applications [27]. The main challenge these systems face is that it is hard to design a type system that is at the same time simple, correct, and amenable to optimizations.

Both scripting and optional types assume that programmers need to restrict the dynamism of their code when they seek better performance. Although in theory JIT compilers do not require this, in practice programmers also need to restrict themselves to achieve maximum performance. Realizing how these self-imposed restrictions result in the creation of vaguely defined language subsets, and how restricting dynamism seems unavoidable, we asked ourselves: what if we accepted the restrictions and defined a new programming language based on them? By focusing on this "well behaved" subset and making it explicit, instead of trying to optimize or type a dynamic language in its full generality, we would be able to drastically simplify the type system and the compiler.

To study this question, we are developing the Pallene programming language, a statically typed companion to the Lua scripting language. The following list of goals guides our design of Pallene:

- Pallene should be amenable to standard ahead-of-time compiler optimization techniques.
- Pallene should share the Lua runtime, using the same data representation and the same garbage collector.
- Pallene should be familiar to Lua programmers, syntactically and semantically.

In the next section of this paper, we introduce the main approaches currently used to tackle the performance problems of dynamic languages. In Section 3, we describe how we designed Pallene, aiming to combine desirable properties from these approaches. In Section 4, we discuss how our goals for Pallene affected its implementation. In Section 5, we evaluate the performance of our prototype implementation of Pallene on a set of micro benchmarks, comparing it with the reference Lua implementation and with LuaJIT [20], a state of the art JIT compiler for Lua, showing that it is possible to produce efficient executable code for Pallene programs. In the last two sections of this paper we compare Pallene with related research in optimizing dynamic languages and type systems for dynamic languages, and we discuss avenues for future work.

2 OPTIMIZING SCRIPTING LANGUAGES

In this section we discuss existing approaches to optimizing dynamic languages that we mentioned in the introduction.

[1] Gualandi [9] presents a historical review of type systems for dynamic languages.

2.1 Scripting

One way to overcome the slowness of dynamic languages is to avoid them for performance-sensitive code. Dynamic scripting languages are often well-suited for a multi-language architecture, where a statically typed low-level *system language* is combined with a flexible dynamically typed *scripting language*, a style of programming that has been championed by John Ousterhout [19].

Lua has been designed from the start with scripting in mind [13]. Many applications that use Lua follow this approach. For instance, a computer game like Grim Fandango has a basic engine, written in C++, that performs physical simulations, graphics rendering, and other machine intensive tasks. The game designers, who are not professional programmers, wrote all the game logic in Lua [18].

The main advantages of this scripting architecture are its pragmatism and its predictability. Each language is used where it is more adequate and the system architect can be relatively confident that the parts written in the system language will have good performance. The main downside is the conceptual mismatch between the languages.

Rewriting modules from one language to the other is difficult. A common piece of advice when a Lua programmer seeks better performance is to "rewrite it in C". This is easier said than done. In practice, programmers only follow this advice when the code is mainly about low-level operations such as array accesses, arithmetic, and calling external libraries and the operating system. Another obstacle to this suggestion is that it is hard to estimate in advance both the costs of rewriting the code and the performance benefits to be achieved by the change.

Often, the gain in performance is not what one would expect. As we will see in Section 5, the overhead of converting data from one runtime to the other can be very high, canceling out much of the inherent gains of switching to a static language.

2.2 JIT Compilers

Just-in-time (JIT) compilers are the state of the art in dynamic language optimization. A JIT compiler initially executes the program without any optimization, observes its behavior at runtime, and based on this generates highly specialized and optimized executable code. For example, if it observes that some code is always operating on values of type *double*, the compiler will optimistically compile a version of this code that is specialized for that type. It will also insert tests (guards) in the beginning of the code that jump back to a less optimized generic version in case some value is not of type *double* as expected.

JIT compilers are broadly classified as method-based or trace-based [7], according to their main unit of compilation. In method-based JITs the unit of compilation is one function or subroutine. In trace-based JITs, the unit of compilation is a linear trace of the program execution, which may cross over function boundaries. Trace compilation allows for a more embedable implementation and is better at compiling across abstraction boundaries. However, trace compilation has difficulty optimizing programs which contain unpredictable branch statements. For this reason, now most JIT compilers tend to use the method-based approach, with the notable exceptions of LuaJIT [20] and the RPython Framework [5].

```
-- Bad
local function mytan(x)
    return math.sin(x) / math.cos(x)
end

-- Good
local sin, cos = math.sin, math.cos
local function mytan(x)
    return sin(x) / cos(x)
end
```

Figure 1: LuaJIT encourages programmers to cache imported Lua functions in local variables.

JIT compilers detect types at run-time because inferring types at compile type is very hard and usually produces less specific results. Additionally, in many dynamic languages data types are created at run-time and there are no data definition declarations or type annotations for the compiler to take advantage from. As an example, the PyPy authors explicitly mentioned this as one of their main motivations for using JIT technology [23].

Implementing a JIT compiler can be challenging. The most performant JITs depend heavily on non-portable low-level code and are architected around language-specific heuristics. High level JIT development frameworks are still an active research area. There are various promising approaches, such as the *metatracing* of the RPython framework [5] and the partial evaluation strategy of Truffle [29], but so far these have not been able to compete in terms of performance and resource usage with hand-written JITs such as LuaJIT [20], V8 [26] and HHVM [1].

From the point of view of the software developer, the most attractive feature of JIT compilers is that they promise increased performance without needing to modify the original dynamically typed program. However, the gains are not always easy to achieve, because the effectiveness of JIT compiler optimizations can be hard to predict. Certain code patterns, known as *optimization killers*, may cause the whole section they are in to be de-optimized, which can result in a dramatic performance impact. Programmers who seek performance must carefully avoid these optimization killers, which are often described in official or unofficial documentation for popular JIT engines [22, 2].

Since there may be an order of magnitude difference in performance between JIT-optimized and unoptimized code, programmers have an incentive to write their programs in a style that is more amenable for optimization. This leads to idioms that are not always intuitive. For example, the LuaJIT documentation recommends caching Lua functions from other modules in a local variable before calling them, as shown in Figure 1 [21]. However, for C functions defined via the foreign function interface the rule is the other way around: functions from the `C` namespace should not be cached in local variables, as shown in Figure 2.

Another example from LuaJIT is the function in Figure 3, which runs into several LuaJIT optimization killers (which are referred by LuaJIT as "Not Yet Implemented" features). As of LuaJIT 2.1, traces that call string pattern-matching methods such as `gsub` are

```
-- Good (!)
local function hello()
    C.printf("Hello, world!")
end

-- Bad (!)
local printf = C.printf
local function hello()
    printf("Hello, world!")
end
```

Figure 2: Surprisingly, LuaJIT encourages programmers not to cache C functions called through the foreign function interface.

```
function increment_numbers(text)
    return (text:gsub("[0-9]+", function(s)
        return tostring(tonumber(s) + 1)
    end))
end
```

Figure 3: This function cannot be optimized by LuaJIT because it calls the `string.gsub` method and because it uses an anonymous callback function.

not compiled into machine code by the JIT. The same is true for traces that create closures or use anonymous functions, even if the anonymous function doesn't close over any outer variables.

Besides writing code in a very restricted style, programmers still need to double check if the JIT compiler is actually optimizing their code. If it is not, they need to debug their programs to find out why, using specialized tools [11]. This may require the programmer to work at a low level of abstraction, involving the intermediate representation of the JIT compiler or its generated machine code [12].

Another aspect of JIT compilers is that before they start optimizing they need to run the program for many iterations, collecting run-time information. During this initial warmup period the JIT will run only as fast or even slower than a non-JIT implementation. In some JIT compilers the warmup time can also be erratic, or even cyclic, as observed by Barret et al [3].

2.3 Optional Types

Static types serve several purposes. They are useful for error detection and as a lightweight documentation, and they facilitate efficient code generation. Therefore there are many projects that aim to combine the benefits of static and dynamic typing in a single language.

A recurring idea to help the compiler produce more efficient code is to allow the programmer to add optional type annotations to the program. Compared with a more traditional scripting approach, optional typing promises a single language instead of two different ones, which makes it easier for the static and dynamic parts of the program to interact with each other. The pros and cons of these optional type system approaches vary from case to case, since each type system is designed for a different purpose. For example, the optional type annotations of Common LISP allow the compiler to generate extremely efficient code, but without any safeguards [8].

A research area deserving special attention is Gradual Typing [24]. It is a promising technique for designing type systems that aim to provide a solid theoretical framework for integrating static and dynamic typing in a single language. However, gradual type systems still face difficulties when it comes to run-time performance. On the one hand, systems that try to check types as they cross the boundary between the static and dynamic parts of the code face a high verification overhead cost [25]. On the other hand, type systems that do not perform this verification give up on being able to optimize the static parts of the program.

One problem with optional types is that, to embrace the ideal of smooth transition between the typed and untyped worlds, the static type system should support common idioms from the dynamic language. This requirement for flexibility usually results in a more complex type system, making it more difficult to use and, more importantly to us, to optimize. For example, in Typed Lua [16] all arrays of integers are actually arrays of nullable integers. In Lua, out of bound accesses result in `nil`; moreover, to remove an element from a list one has to assign `nil` to its position. Both cases require the type system to accept `nil` as a valid element of the list.

3 THE PALLENE PROGRAMMING LANGUAGE

Although JIT compilers and optional type systems are said to be designed to cover all aspects of their dynamic languages, this is not the case in practice. Normally there is a "well behaved" subset of the language that is more suitable to the optimizer or the type system, and programmers will restrict themselves to this subset to better take advantage of their tools.

For example, programmers targeting LuaJIT will tend to restrict themselves to the subset of Lua that LuaJIT can optimize. Similarly, those using TypeScript (an optional type system for Javascript) will prefer to write programs that can fit inside TypeScript's type system. From a certain point of view, we could say that these programmers are no longer programming in Lua or Javascript, at least as these languages are normally used.

As we already mentioned in the introduction, the realization that programmers naturally restrict themselves to a subset of the language in the search for better performance led us to think about a new language that makes these restrictions explicit. Our hypothesis is that programmers would be willing to accept these restrictions in exchange for guarantees from the compiler that it will be able to generate good code.

Pallene is intended to be that language. As we described before, our goals for Pallene were to make it amenable to standard ahead-of-time compiler optimization techniques and compatible with Lua, not only in terms of the run time and data structures, but also in terms of language semantics and familiarity. In the following paragraphs we describe how these goals affected the design of Pallene.

Pallene should be amenable to standard ahead-of-time compiler optimization techniques. This goal led us to make Pallene statically

```
function sum(xs: {float}): float
  local s: float = 0.0
  for i = 1, #xs do
    s = s + xs[i]
  end
  return s
end
```

Figure 4: A Pallene function for summing numbers in a Lua array.

```
-- Pallene Code:
function add(x: float, y:float): float
    return x + y
end

-- Lua Code:
local big = 18014398509481984
print(add(big, 1) == big)
```

Figure 5: An example of problems with coercion in Pallene.

typed. Static types make it much easier for a compiler to produce efficient code. The programmer can also rely on the compiler to always generate good code. Pallene only supports language mechanisms for which it can generate good code. This avoids the situation that is common with JIT users, who may need to resort to trial and error to optimize their programs. As an example, Pallene does not support monkey patching: If a piece a code calls **`math.sin`**, Pallene assumes that the function has not been redefined and generates code that directly calls the sine function from the C standard library. If the programmer wants flexibility, they can still use Lua as a scripting language; if they want performance, they might appreciate that Pallene rejects upfront code patterns that it cannot compile effectively.

Pallene should share the Lua runtime. Pallene is able to directly manipulate Lua data structures and call Lua functions. It also shares Lua's garbage collector. This allows a more seamless integration with Lua than is possible for other static languages such as C. These languages must use the Lua's C API to call functions and can only reference Lua values indirectly, through the API's stack and registry [14].

C code, when manipulating Lua values, does not keep direct pointers to Lua objects. Instead, it uses a stack, known as *the Lua stack*. The stack keeps all objects being manipulated by the function, and the function uses integer indices into the stack to refer to these values. The stack facilitates accurate garbage collection (only the objects in the stack are being used) and dynamic typing (stack slots can contain Lua values of any type).

Pallene should be familiar to Lua programmers, syntactically and semantically. To make it easier to combine Lua and Pallene in a single system, we aimed to reuse as many familiar Lua concepts as possible in Pallene. So we aimed to make Pallene as close as possible to a subset of Lua. For example, Figure 4 shows a Pallene function that computes the sum of an array of numbers. Except for the type annotations, this is valid Lua code. Semantically, it also behaves exactly like the Lua version, except that this Pallene function is specialized for floating-point numbers, and will raise a run-time type error if Lua calls it with a different type of parameter (for example, an array of integers). This follows the idea behind the Gradual Guarantee [24] of gradual type systems; it states that adding type annotations to a program should not change its behavior except for perhaps introducing run-time type errors. This guarantee is intended to ease the transition from untyped to typed code.

Whenever reasonable, we try to ensure that the semantics of Pallene is the same as Lua's. For instance, Pallene does not perform automatic coercions between numeric types, unlike most statically typed languages. Consider the example in Figure 5. In Lua, as in many dynamic languages, the addition of two integers in an integer addition. If we remove the type annotations from the Pallene function add, and treat it as Lua code, Lua will perform an integer addition and the program will print **`false`**. If Pallene automatically coerced integers to floats, following the type annotation of the parameters, it would perform a floating-point addition and the program would print **`true`**: double-precision floating-point numbers cannot accurately represent $2^{54} + 1$. To avoid this inconsistency, Pallene instead raises a run-time type error, complaining that an integer was passed where a floating-point value was expected.

Nevertheless, sometimes breaking strict compatibility can be desirable to achieve better performance and keep the language simple. A good example is how Pallene ignores monkey-patching. Making Pallene's module system less dynamic opens up many possibilities for optimization. At the same time, programmers concerned with performance most likely already avoid this feature and won't notice that it is missing in Pallene.

4 IMPLEMENTING PALLENE

The Pallene compiler compiles Pallene to C and then calls a C compiler (such as gcc) to produce the executable code. The code that Pallene generates complies with Lua's Application Binary Interface (ABI) so that it can be dynamically loaded by Lua just like other modules written in C.

Leveraging an existing high quality compiler backend keeps the Pallene compiler simple and makes Pallene portable to many architectures. Using C also allows us to reference datatypes and macros defined in Lua's C header files.

Conventional C libraries for Lua use a well-defined API to interact with the Lua interpreter [14]. Pallene instead directly accesses the internals of the Lua interpreter and Pallene code can hold pointers to Lua data structures, which isn't allowed for regular C code. This allows better performance than what is currently possible with C Lua modules as we will show. This style of programming would be dangerous if exposed to C programmers, but in Pallene the compiler is able to guarantee that the invariants of the Lua interpreter are respected.

Pallene's low-level approach is more similar to the API of other scripting languages such as Python or Ruby's. Unlike with Python,

Pallene programmers don't need to perform manual memory management. Pallene's GC is accurate, unlike Ruby which is conservative regarding local variables in the C stack.

Pallene currently uses *lazy pointer stacks* [15] to interact with the garbage collector. Collectable Lua values in Pallene are represented as regular C pointers, which at runtime will be stored in machine registers and the C stack, with low overhead. At points in the program where the garbage collector is invoked, Pallene saves all the necessary Lua pointers in the Lua stack, so that the garbage collector can see them.

Pallene must insert run-time type checks in the frontier between its statically typed code and Lua's dynamically typed code, in order to guarantee type safety. Currently, this means function calls (when a Lua function calls a Pallene one) and data-structure reads (Lua may write values of the wrong type to the field).

We avoided using wrappers to implement our type tests because they would not interact well with the reference equality (object identity) of Lua and Pallene. Obtaining good performance in a system with wrappers is also challenging. In the worst case they can even slow down the whole program by a factor of more than ten, as shown by Takikawa et al [25]; avoiding this overhead is still an open problem. Instead of wrappers, we use a more lazy system of type checks, similar to the tag checks inserted by JIT compilers to guard their specialized code or to the *transient semantics* for Reticulated Python [28].

5 PERFORMANCE EVALUATION

We want to verify whether Pallene programs can compete with JIT compilers and we want to verify whether bypassing the Lua–C API can lead to relevant performance gains.

We prepared a small suite of benchmarks to evaluate the performance of Pallene. The first benchmark is a prime sieve algorithm. The second is a matrix multiplication (using Lua arrays). The third one solves the N-queens problem. The fourth one is an microbenchmark that simulates a binary search. The fifth one is a cellular automaton simulation for Conway's Game of Life.

All the benchmarks were prepared specifically for this study. Except for the binary search, all other algorithms are superlinear with the input size, so it was easy to choose an appropriately large N. For the binary search, we resorted to repeating the computation inside a loop. As expected, all benchmarks run the same in Pallene and Lua: the source code is identical, except for type annotations, and they produce the same results. Most of the code is what one would naturally write in Lua (except for the type annotations). One exception was in the code for the N-queens benchmark, where we used if-then-else statements in places where in Lua it would be more idiomatic to use the idiom `x` **`and`** `y` **`or`** `z` as a ternary operator. (Currently Pallene only supports using **`and`** and **`or`** with boolean operands.) For the matrix multiplication and the cellular automaton benchmarks, we manually hoisted some loop-invariant operations. Lua programmers often do that when they are concerned about performance. LuaJIT already implements this optimization, but Pallene doesn't yet. By performing it manually, we could obtain a more accurate comparison of the costs of array read and write operations.

We also implemented all these benchmarks in C, but using Lua data structures manipulated through the Lua–C API. This allowed us to compare how the scripting approach compares to Pallene, which bypasses the API.

We ran our experiments in a computer with a 3.20 GHz Ivy Bridge Intel CPU and normalized the execution times to the ones of the reference Lua implementation (PUC-Lua). We used Lua version 5.4-work1 for the benchmarks.[2]

We compared Pallene programs with their Lua equivalents, ran under both the reference Lua implementation and under LuaJIT. In all benchmarks the only difference between the Lua and Pallene programs is the presence of type annotations, since we restricted the Lua programs to the language subset supported by Pallene. The results are shown in Figure 6.

The first benchmarks—prime sieve and matrix multiplication—heavily feature array operations and arithmetic. In both, Pallene and LuaJIT achieved similar performance, with an order of magnitude improvement when compared to the reference Lua implementation. The Lua–C API implementation had much smaller gains. This shows the costs of using the Lua–C API at this level of granularity (single access to array elements). The small gain is probably due to arithmetic being performed in C instead of in dynamically-typed Lua.

The third benchmark—the N-queens problem—also features array operations and arithmetic, but has a larger proportion of arithmetic compared to array operations. Pallene and LuaJIT performed as well as they did in the previous two benchmarks. The C-API did better, due to the heavier weight of arithmetic operations.

In the fourth benchmark—binary search—Pallene ran more than three times faster than LuaJIT and the Lua–C API implementation. We can see that the performance of Pallene was similar to the other benchmarks which indicates that the difference is due to LuaJIT doing worse on this particular benchmark. The binary search does many unpredictable branches, which is very bad for trace-based JIT compilers. This illustrates the unpredictability of JIT compilation in general and trace-based JIT compilation in particular.

The final benchmark—Conway's game for life—spends much time doing string operations and generates a lot of garbage. The C-API implementation could not obtain a speedup compared to the reference interpreter and LuaJIT could barely beat it. We suspect that this is due to recent improvements in the PUC-Lua garbage collector. (Lua 5.3 takes 150% longer to run than 5.4.)[3] Given that this benchmark spends so much time in the garbage collector, Pallene's 2x speedup seems respectable.

6 RELATED WORK

JIT compilers answer a popular demand to speed up dynamic programs without the need to rewrite them in another language. However, they do not evenly optimize all idioms of the language, which in practice affects programming style, encouraging programmers to restrict themselves to the optimized subset of the language. Pallene has chosen to be more transparent about what can be optimized, and made these restrictions a part of the language.

[2]The source code for the Pallene compiler and the benchmarks can be found at https://github.com/titan-lang/titan/releases/tag/sblp

[3]LuaJIT had plans to update its garbage collector but that still hasn't happened.

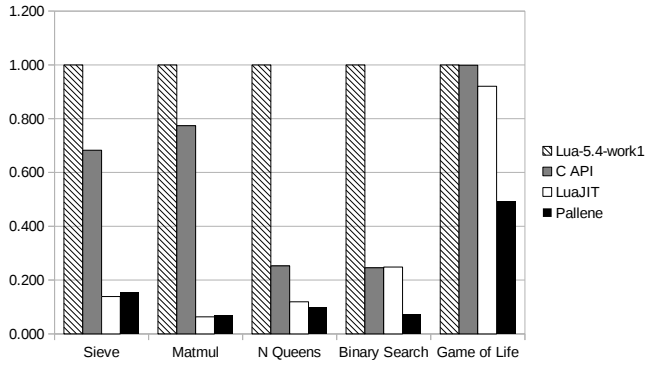

Figure 6: Performance evaluation of Pallene.

Like Gradual Typing systems, Pallene recognizes that adding static types to a dynamic language provides many benefits, such as safety, documentation, and performance. Pallene is also inspired by the Gradual Guarantee, which states that the typed subset of the language should behave exactly as the dynamic language, for improved interoperability. Unlike many gradually typed systems, Pallene can only statically type a restricted subset of Lua. This avoids the complexity and performance challenges that are common in many gradually typed systems.

Common Lisp is another language that has used optional type annotations to provide better performance. As said by Paul Graham in his ANSI Common Lisp Book [8], "Lisp is really two languages: a language for writing fast programs and a language for writing programs fast". Pallene and Common Lisp differ in how their sublanguages are connected. In Common Lisp, they live together under the Lisp umbrella, while with Pallene they are segregated, under the assumption that modules can be written in different languages.

Cython [4] is an extension of Python with C datatypes. It is well suited for interfacing with C libraries and for numerical computation, but its type system cannot describe Python types. Cython is unable to provide large speedups for programs that spend most of their time operating on Python data structures.

Terra is a low-level system language that is embedded in and meta-programmed by Lua. Similarly to Pallene, Terra is also focused on performance and has a syntax that is very similar to Lua, to make the combination of languages more pleasant to use. However, while Pallene is intended for applications that use Lua as a scripting language, Terra is a stage-programming tool. The Terra system uses Lua to generate Terra programs aimed at high-performance numerical computation. Once produced, these programs run independently of Lua. Terra uses manual memory management and features low-level C-like datatypes. There are no language features to aid in interacting with a scripting language at run-time.

7 CONCLUSION AND FUTURE WORK

Our initial results suggest that Pallene's approach of providing a statically typed companion language for a dynamically typed scripting language is a promising approach for applications and libraries that seek good performance. It apparently will be able to compete with LuaJIT as an alternative to speeding up Lua applications. As we expected, Pallene performs better than the scripting approach. Moreover, porting a piece of code from Lua to Pallene seems much easier than porting it to C.

We also want to study the impact of implementing traditional compiler optimizations such as common sub-expression elimination and loop invariant code motion. Our current implementation relies on an underlying C compiler for almost all optimizations, and our work suggests that implementing some optimizations at the Pallene level might lead to significant improvements. The C compiled cannot be expected to understand the Lua-level abstractions needed to perform these optimizations.

One question we wish to answer in the future is whether the type system simplicity and the good performance results we achieved in the array-based benchmarks will be preserved as we add more language features, such as records, objects, modules and a foreign function interface.

8 ACKNOWLEDGMENTS

We would like to thank Hisham Muhammad, Gabriel Ligneul, Fábio Mascarenhas, and André Maidl for useful discussions about the initial ideas behind Pallene. We would also like to thank Sérgio Medeiros for assisting us with the developement of Pallene's scanner and parser.

Pallene was born as a fork of the Titan [17] programming language, with a focus on researching the performance aspects of dynamic programming language implementation. Gabriel Ligneul heavily contributed to current implementation of the Pallene compiler.

REFERENCES

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. 777–790. DOI: http://dx.doi.org/10.1145/2660193.2660199

[2] Petka Antonov and others. 2013. V8 Optimization Killers. (2013). https://github.com/petkaantonov/bluebird/wiki/Optimization-killers Retrieved in 2017-01-08. Full author list available at https://github.com/petkaantonov/bluebird/wiki/Optimization-killers/_history.

[3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. In *Proceedings of the 32nd Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '17)*. DOI: http://dx.doi.org/10.1145/3133876

[4] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 2 (March 2011), 31–39. DOI: http://dx.doi.org/10.1109/MCSE.2010.118

[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, 18–25. DOI: http://dx.doi.org/10.1145/1565824.1565827

[6] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. DOI: http://dx.doi.org/10.1145/800017.800542

[7] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. New York, NY, USA, 144–153. DOI: http://dx.doi.org/10.1145/1134760.1134780

[8] Paul Graham. 1996. *ANSI Common LISP*. Apt, Alan R. http://www.paulgraham.com/acl.html

[9] Hugo Musso Gualandi. 2015. *Typing Dynamic Languages – a Review*. M.S. thesis, Pontificia Universidade Católica do Rio de Janeiro (PUC-Rio).

[10] Javier Guerra. 2017. LuaJIT Hacking: Getting next() out of the NYI list. CloudFare Blog. (Feb. 2017). https://blog.cloudflare.com/luajit-hacking-getting-next-out-of-the-nyi-list/.

[11] Javier Guerra Giraldez. 2016. LOOM - A LuaJIT performance visualizer. (2016). https://github.com/cloudflare/loom

[12] Javier Guerra Giraldez. 2017. The Rocky Road to MCode. Talk at Lua Moscow conference, 2017. (2017). https://www.youtube.com/watch?v=sz2CuDpltmM

[13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 2–1–2–26. DOI: http://dx.doi.org/10.1145/1238844.1238846

[14] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2011. Passing a Language Through the Eye of a Needle. *Commun. ACM* 54, 7 (July 2011), 38–43. DOI: http://dx.doi.org/10.1145/1965724.1965739

[15] Baker J., Cunei A., Kalibera T., Pizlo F., and Vitek J. 2009. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience* 21, 12 (2009), 1572–1606. DOI: http://dx.doi.org/10.1002/cpe.1391

[16] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2015. A Formalization of Typed Lua. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 13–25. DOI: http://dx.doi.org/10.1145/2816707.2816709

[17] André Murbach Maidl, Fábio Mascarenhas, Gabriel Ligneul, Hisham Muhammad, and Hugo Musso Gualandi. Source code repository for the Titan programming language. (????). https://github.com/titan-lang/titan

[18] Bret Mogilefsky. 1999. Lua in Grim Fandango. Grim Fandango Network. (May 1999). https://www.grimfandango.net/features/articles/lua-in-grim-fandango.

[19] John K. Ousterhout. 1998. Scripting: Higher-Level Programming for the 21st Century. *Computer* 31, 3 (March 1998), 23–30. DOI: http://dx.doi.org/10.1109/2.660187

[20] Mike Pall. 2005. LuaJIT, a Just-In-Time Compiler for Lua. (2005). http://luajit.org/luajit.html

[21] Mike Pall. 2012. LUAJIT performance tips. lua-l mailing list. (nov 2012). http://wiki.luajit.org/Numerical-Computing-Performance-Guide

[22] Mike Pall and others. 2014. Not Yet Implemented operations n LuaJIT. LuaJIT documentation Wiki. (2014). http://wiki.luajit.org/NYI Retrieved 2017-01-08. Full author list available at http://wiki.luajit.org/history/NYI.

[23] Armin Rigo, Michael Hudson, and Samuele Pedroni. 2005. *Compiling Dynamic Language Implementations*. Tech. rep., Heinrich-Heine-Universität Düsseldorf. https://bitbucket.org/pypy/extradoc/raw/tip/eu-report/D05.1_Publish_on_translating_a_very-high-level_description.pdf

[24] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL '2015)*. Asilomar, California, USA, 274–293. DOI: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.274

[25] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 456–468. DOI: http://dx.doi.org/10.1145/2837614.2837630

[26] The Chromium Project. 2008. The Chrome V8 Engine. (2008). https://developers.google.com/v8/ Retrieved 2017-01-08.

[27] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 964–974. DOI: http://dx.doi.org/10.1145/1176617.1176755

[28] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 45–56. DOI: http://dx.doi.org/10.1145/2661088.2661101

[29] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. 187–204. DOI: http://dx.doi.org/10.1145/2509578.2509581