Evaluating Optimizations for a High-Level Language

Leonardo Kaplan
lkaplan@inf.puc-rio.br
PUC-Rio
Brazil

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

ABSTRACT

Pallene aims to be a system language counterpart for Lua, with similar syntax but ahead-of-time compilation. It also has optional typing and stricter semantics, allowing it to emit C code with unboxed values, which enable several optimizations in the C compiler. Assuming that, the current Pallene compiler does not implement most classic optimizations. However, the C compiler cannot perform all the expected optimizations due to the difference in the level of abstraction between Pallene and C. In this work, we studied several scenarios where the C compiler couldn't perform those expected optimizations trying to understand why and implemented them in the Pallene compiler, which has higher-level understanding of the language. Among the implemented optimizations are function inlining and scalar replacement of aggregates, achieving results that bring evidence that higher level optimizations cannot be done in the lower level compiler.

CCS CONCEPTS

• **Software and its engineering → Dynamic compilers**.

ACM Reference Format:
Leonardo Kaplan and Roberto Ierusalimschy. 2021. Evaluating Optimizations for a High-Level Language. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3475061.3475088

1 INTRODUCTION

Dynamic languages provide flexibility and simplicity in exchange for less compile-time information, leading to slower run times. While we can reduce the performance impact of using dynamic languages with scripting, JITs or optional typing [9], neither solution is perfect: Scripting may require extensive rewrites and the API may introduce overhead. JITs' implementations are usually complex and non-portable and with unpredictable performance killers. Optional typing doesn't have a tradition of being used for optimization.

This problem was explored in the context of PHP[1] and Java[13], with approaches similar to the ones described in this work.

Addressing this problem in the Lua context, the Pallene programming language [8] appears as an alternative. Characterized as a companion language for Lua, it combines the scripting approach with optional typing to create a language that acts as a system language while sharing the run-time with its dynamic counterpart. The type system provides run-time checks for data coming from Lua to Pallene and static checks on the other direction. The shared run-time in the place of an API significantly reduces the data communication overhead.

The Pallene compiler generates C code from Pallene code. A conventional C compiler then compiles the C code into a library, which can then be loaded by the Lua interpreter. The loaded code interacts directly with the Lua world, without the need of the standard Lua API.

Pallene can generate efficient C code because of its type system. Pallene's type system gives the compiler enough information to represent values unboxed, so we can emit more efficient code. Moreover, Pallene uses run-time checks to determine whether a value coming from Lua is well-typed (in respect to its annotations) or whether it is a run-time error. Besides that, Pallene also forbids almost all of Lua's dynamic behaviors, such as monkey patching, polymorphic functions, and metamethods. These guarantees enables the generation of code specialized for its types.

Because the Pallene compiler generates C, it doesn't need to implement classic compiler optimizations. Conventional C compilers usually implement a wide range of optimizations that can or cannot be applied depending on code to be compiled. With unboxed values and with the before-mentioned stricter semantics, the code emitted by Pallene usually can take for granted the already implemented optimizations in the C compiler.

While delegating all optimizations to the lower level compiler would be the ideal case, it isn't always possible: Pallene and C operate at different levels of abstraction, and as a result, simple Pallene operations end up appearing to C as unpredictable side-effects.

The difference of abstraction level makes some optimizations impossible or at least makes their implementation impractical. An example is to fold constants while trying to account for Lua's emergency garbage collector's behavior: The collector acts when memory allocation fails, doing a full collection cycle to then retry the failed allocation. For Pallene, these emergency calls are invisible, while for C, each can be seen as a complete state change.

In this work, we will analyze possible opportunities for optimizations in the Pallene compiler. These opportunities will be extracted from our study on the current state of the compilation pipeline (from Pallene code to C and then object code). We will then propose some optimizations, describe their implementations and their results. Achieving significant results is an evidence that there are optimizations that should be done at the Pallene compiler. We believe that this result can be generalized for systems that compile from a higher level language to a lower level.

SBLP'21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9062-0/21/09…$15.00
https://doi.org/10.1145/3475061.3475088

2 THE DESIGN OF PALLENE

The main goal of Pallene is to be seamlessly interoperable with Lua while being predictably efficient with a simple, portable implementation.

Pallene is a statically typed subset of Lua. It guarantees that in removing the type annotations, one always end up with a valid Lua program with the same semantics. We can see an example of the type annotations on Listing 1 and 2. All parameters and return values must be explicitly typed, but the type annotations for local variables with constant values can be omitted, as we see in line 3 of the mentioned code.

Pallene is a compiled ahead-of-time system language. A system language is the counterpart of the application language in a scripting system. While the application language is built for flexibility and ease of use, the system language is designed to achieve greater performance. The system language usually achieves greater performance than the application language by operating in a lower level of abstraction than its counterpart.

Other system languages need to use the Lua-C API to interact with Lua, while Pallene can directly manipulate Lua's values. The API exists to provide some guarantees about invariants of Lua's data structures, but the Pallene compiler has all the necessary information on Lua's internal data structures, so it can manipulate them directly and safely, effectively using Lua's runtime. In fact, Pallene creates all of its data structures inside Lua's runtime, which allows them to be tracked by Lua's garbage collector.

The Pallene compiler emits C code. The emitted code uses the same implementation tradition that Lua uses, aiming to be as portable as Lua. The C code can be compiled to a module with any conventional C compiler. After a Pallene module is compiled, it can be required and used in Lua as a regular C module. The code that the Pallene compiler emits, being C, can take for granted several optimizations that most modern C compilers implement. One effect of this is that the Pallene compiler itself doesn't need to implement traditional optimizations.

```
1 function foo(N: integer): integer
2     local a: integer = 10
3     local b = 20
4     return N + a + b
5 end
```

Listing 1: Example of a Pallene program that receives an integer, add it to constants and returns the result

```
1 function foo(N)
2     local a = 10
3     local b = 20
4     return N + a + b
5 end
```

Listing 2: Example of a Lua program that receives an integer, add it to constants and returns the result

3 OPTIMIZATIONS DURING C COMPILATION

Because Pallene's design leaves most of the optimizations for the lower level compiler, our work on optimizing the Pallene compiler starts with defining a baseline of what a C compiler can or cannot optimize. One way of doing this is to compile samples of C code to assembly and check which optimizations were performed.

In this study, we used the GCC compiler, version 9.3.0. We used the O2 optimization mode to generate an assembly with commentaries.

We aggregated some Pallene code samples and examined the assembly of each. We compiled each sample to C using the Pallene compiler and then to assembly using the C compiler. In Listing 3 we can examine an example of Pallene code that does some basic arithmetic and assignment.

```
1 function f() : integer
2     local x = 43
3     local y = 71
4     local z = x
5     return z + y
6 end
```

Listing 3: An example of sum of constants and assignments in Pallene

When we compile Listing 3 to C, the code remains mostly equal, as expected. The C compiler, on the other hand, transforms the function call into a single constant, 114. This optimization, called constant folding [11] or constant propagation [14], substitutes variables with their values, when the values can be known in compile time. The C compiler can perform this optimization on normal C integers [5]. Because of that, it can also optimize Pallene's local integers, which are emitted as simple unboxed C integers.

The C compiler can fold heap values as well. In Listing 4, we changed the local variable x to be an array, modifying lines 2 and 4 accordingly. The C compiler still can fold the whole function call into a constant. Even with the array being created in Lua's runtime, it is still visible to the C compiler as a conventional array. Since the first value of the array doesn't change, it can perform the scalar replacement of aggregates optimization [10] (controlled by the flags fipa–sra and ftree–sra, for local and interprocedural scopes, respectively). This optimization transforms components of an aggregate (an array, for instance) into scalar variables (local integers, in this case), effectively making the programs in Listings 3 and 4 the same.

```
1 function f() : integer
2     local x: {integer} = {43}
3     local y = 71
4     local z = x[1]
5     return z + y
6 end
```

Listing 4: Sum of heap constants in Pallene

```
1 function f() : integer
2     local x:{integer} = {43}
3     local y = 71
4     local s = 'a'..'b'
5     local z = x[1]
6     return z + y
7 end
```

Listing 5: Sum of heap constants in Pallene with string concatenation

Although the C compiler can fold both stack and heap values, there are still cases in which it can't fold properly. By inserting a string concatenation after line 3 in the code 4 (resulting in Listing 5), we can disable the before-mentioned optimization. To the Pallene user, this code shouldn't have any interference with the rest of the function, still it makes his code slower. The existence of code patterns that surprisingly worsen performance goes against the design goal of Pallene of having predictable efficiency.

The transference of control to Lua disabled the optimization. String concatenation in Pallene is one of the built-in functions that transfer the control to Lua. At the C compiler level, this transference of control makes some of the tracked information to be invalidated. In particular, it loses the guarantee that the array hasn't changed. We as Pallene users can know that because of the semantics of the concatenation operator, which doesn't have any side-effect that could change the array, but this knowledge is too abstract for the C compiler.

The particular problem described for Listing 5 can be solved using the scalar replacement of aggregates (SRA) optimization [2] at the Pallene compiler. The optimization aims to substitute arrays accesses for their respective values when possible, possibly eliminating the whole array if its uses were all replaced. We could replace the `x[1]` for its value, 43, making it possible to eliminate its creation. This makes the runtime of the program to be similar with the one described in Listing 3.

It is easy do determine the scalar equivalent of the index because the array creation was in context. When the array is received as a parameter, it becomes significantly harder to infer its values. One form of having both the array creation and its uses in the same context is by inlining the function calls. At the same time, this strategy only works when the functions are all written in Pallene (not in Lua), in special the function that creates the array. This kind of situation, in which inlining enables other optimizations became a motivation for the implementation of function inline at the higher level compiler. At the same time, the optimization itself can be useful for providing inlining where the lower level compiler can't detect its possibility.

If we examine the generated C code for Listing 5, we will find several function calls that weren't present in the Pallene program. They perform checks that are simply invisible for the higher abstraction level. In particular, there is the array normalization check, that verifies if a given array has a given size. Since checking for an index gives us the guarantee that the array has at least the index size, we can propose an optimization (array renormalization): We can detect the upper-bound of these sizes to be checked, reducing the total amount of checks, only checking for the upper bounds.

In order to evaluate these three proposed optimizations, SRA, function inline and the reduction in array normalization checks, we compiled the Pallene code of some benchmarks to C and applied the optimizations by hand in each one, as the compiler would omit them. Verifying that they had significant impact, we proceeded to their proper implementation.

The array renormalization and the SRA optimizations require context information to be performed. Both need escape analysis[3], while SRA also needs some alias analysis to track the aggregates' fields and renormalize needs range analysis to determine the upper bounds for each array size-check. We opted to gather this information with the use of abstract interpretation.

Abstract Interpretation is a form of gathering information of the code. The information that can be gathered usually revolves around the range of possible values that each variable may have in each point of the program. In particular, we can analyse whether this range of values can be known or whether this range has only one possible value. This information can then be used to produce reports (such as warnings or errors) and to allow compiler transformations, as in our case.

We could obtain precise information of the variables' possible values by interpreting the code concretely. The problem with this approach is that it is not computable, this follows from Rice's theorem [12]. At the same time, with abstract interpretation[4], we can be sure that the interpretation will converge. It can provide this guarantee by giving less precise information on the result. We can do that by admitting a wider range of possibilities for a result.

It is important to note that our implementation of abstract interpretation always follows the same order when traversing the program representation, for example, in a `if` statement, it always checks the condition, then the `if` branch and lastly the `else` branch. This order is particularly useful for identifying each program point. This indexing allow us to represent the states of the program tree as an array. Throughout our discussion of the algorithms' implementation, we will be using this order to describe some of our procedures.

4 EVALUATING THE IMPLEMENTATIONS

To measure the impact of each optimization, we made a set of benchmarks based on the ones used in other works involving Pallene [6][8][7]. We described each briefly. All benchmarks receive parameters. We adjusted these parameters to make sure that the run time would take significant time. This way, we reduce the noise on the final result. We used one second of run time as a minimum, but some benchmarks would take several seconds to run even with the smallest inputs.

We measured the running time of these benchmarks on a laptop computer with a 1.60 GHz Intel Core i5-10210 processor and 8 GB of RAM, running Ubuntu Linux 20.04. The interpreters and compilers used were: 5.3.3 for the reference Lua interpreter and 2.1.0-beta3 for LuaJIT. The C compiler used was GCC 9.3, using the O2 optimization mode. Each benchmark was run 10 times using the perf program. We used perf not only to account for the standard deviation but also to make sure that no performance counters were

abnormal. The results are presented with average time for each benchmark. We calculate gain with the difference between average runtime divided by the control time.

The selected benchmarks were the following:

(1) **binarytrees:** it receives a positive integer N and builds a binary tree of height N, forming $2^{(N+1)} - 1$ nodes (using arrays). Then, a second loop recreates and rechecks recursively the tree, for sequential heights, up to N.
(2) **record-binarytrees:** it is similar to the binarytrees benchmark, but represent the tree nodes as records with two fields, instead of arrays.
(3) **binsearch:** it performs a simple binary search in a sequential array with the size passed as a parameter.
(4) **centroid:** it creates N points and finds the centroid between them.
(5) **record-centroid:** it is similar to the centroid benchmark. However each body is represented with a record with two fields (representing its two-dimensional position) instead of using an array with two slots.
(6) **conway:** simulates the Conway's game of life in a 40 by 80 grid, for a given number of steps.
(7) **fannkuch-redux:** given a positive integer N, it builds a sequential array of 1 to N. Then, for each of the $N!$ permutations, it does a procedure in which the elements of the vector are swapped in position until the first position contains the number 1.
(8) **fasta:** it calculates the similarity between strings representing DNA sequences.
(9) **mandelbrot:** it generates the mandelbrot set of a given integer, N, outputing an image in the netpbm format.
(10) **objmandelbrot:** it is similar to the mandelbrot benchmark but uses arrays instead of pairs of integers.
(11) **record-objmandelbrot:** it computes the same as the objmandelbrot benchmark but represents the points as a record with two fields, instead of an array.
(12) **matmul:** it multiplies a $N \times N$ matrix with itself a number of times.
(13) **nbody:** it computes the solution to the n-body problem. The problem consists of finding the trajectories of astronomical bodies with enough gravity between them to alter the routes. It simulates five bodies for a given number of steps.
(14) **record-nbody:** it similar to nbody, but uses records insteads of arrays to represent the positions and velocities.
(15) **queen:** computes the solution to the N-queens problem, in which we try to place the maximum number of chess queens safely in a $N \times N$ board, with N being a given input.
(16) **sieve:** it computes the sieve of Eratosthenes for a given positive integer N, calculating all the prime numbers up to N.
(17) **spectralnorm:** it calculates the spectral norm of a $N \times N$ square matrix for a given N.

5 IMPLEMENTATION AND RESULTS

5.1 Function Inlining

Function inlining is the process of copying a function body in its calls, adjusting its variables, control flow and side-effects accordingly. The most direct effect of this optimization is to reduce the overhead of calling a function. It also simplifies the call graph, which helps the C compiler to optimize the code further on, specially in the cases that it can't determine whether a Pallene function can be inlined. A very interesting result of this optimization is that the transformed code can have significant more optimization opportunities than that the code before having its function calls inlined.

It is important to keep in mind that we cannot delete the original functions, even if all of their calls were inlined. Pallene functions should be available for Lua to call.

This optimization can be useful in two directions: from the point of view of the caller function and from the point of view of the callee function. From the point of view of the caller, function calls can hinder possible optimizations, so removing them can help the optimizer. On the other side, from the point of view of the callee, many optimizations could be performed with more knowledge of its parameters. When we substitute the call for the body, we can possibly make this information available to the pasted body. Note that this doesn't optimize the function itself, but can provide optimizations for its inlined body.

A drawback of this optimization is that it increases the size of the resulting C code, since it basically duplicates code. This could introduce overheads associated with instruction fetching. However, we have verified that in many cases, C modules with and without the function inline optimization applied would produce assemblies with similar sizes. This is an evidence that this overhead doesn't bring much impact.

Our algorithm has two main steps: forming a call graph of the functions of a module and substituting calls with the corresponding function bodies. Even though we talked about abstract interpretation earlier, our implementation for this optimization does not use it.

In the first step, we build an acyclic call graph. The call graph is a potentially cyclic graph in which the nodes are functions and edges are function calls. We detect the cycles using a simple depth-first search, and when we find a cycle, we break it by order of appearance in the module. That is, if a function A is declared before B and they both call each other, there will be a dependency from B to A, but not on the other way around. The resulting graph is then acyclic.

We then start to inline the calls from the sinks to the sources. Inside a node from a call graph (a function body), we select the calls to be inlined in the order of appearance. One last consideration is when the function call appears inside a more complicated expression. This isn't really an issue, because of the internal representation of the language, which already extracts calls inside expressions into local variables.

For recursive functions, we just inline the first level of recursion. The subsequent calls are left as function calls and marked as non-inlinable.

A considerable part of the algorithm is on adjusting the inner variables, parameters and returns to not conflict with the caller function.

Another situation to consider is when a return command is present inside a loop, and more specifically, a nested loop. If a return value is found in the code, we simply replace the values it is returning for the appropriate local variables. This is simple, even considering the multiple returns that Lua and Pallene implement.

Benchmark	Control run time avg. (sec)	Number of calls inlined	Difference in run time (%)
binarytrees	3.63	6	18.18
recordbinarytrees	4.86	6	12.14
binsearch	4.82	1	2.07
centroid	1.51	0	0
recordcentroid	2.20	0	0
conway	4.80	6	12.08
fannkuchredux	4.82	0	0
fasta	2.87	3	9.06
mandelbrot	7.77	0	0
matmul	9.67	0	0
nbody	3.35	1	4.18
recordnbody	8.30	1	4.10
objmandelbrot	9.83	6	6.00
recordobjmandelbrot	14.09	6	6.03
queen	2.30	3	9.13
sieve	28.94	0	0
spectralnorm	27.19	4	8.02

Table 1: Performance gains of performing the function-inlining optimization

When the return is found inside a loop, we must add a break command, in order to maintain the same semantics for control flow. However, break just breaks the first enclosing loop. If there are nested loops, we would have to add control variables and conditions to properly make this transformation. Another option was to introduce goto to the language, breaking its control flow structure and making other transformations harder or even impossible. We then opted on implementing a simple version of named scopes, referencing each scope with a name and breaking the flow to a given reference. Given these labels, it is easier to transform these situations. This was implemented using annotations on the internal representation. These annotations are introduced, used and removed inside the function-inline module, making them invisible to the rest of the compiler.

After all possible function calls of a module are finished being inlined, a last step of normalization for the whole module is performed, specially to guarantee that all variables identifiers are pointing to their corresponding information (name, type and debug information) in the function's variable information array.

For most cases, the optimization wasn't applied or hadn't significant results (less than 5%). But for the binarytrees, record-binarytrees and conway, it showed results between 12% and 18%. In these cases, the inlined function were called repeatedly (exponentially to the input) and the inlined function body were quite small, with a few expressions. Fasta, queen and spectralnorm yielded results close to 10%. In these cases, the inlined functions were also small and called repeatedly, but linearly. We can see all results on Table 1.

5.2 Scalar replacement of arrays

Handling memory is one of the most costly operations in Pallene: Not only the path to the RAM takes more time than the path to processor's caches, it can't be easily optimized by the C compiler. Since the main purpose of Pallene is to serve as a system counterpart to Lua, we can assume that using arrays (or records) to handle groups of data is a common pattern. The optimization is to replace an N-sized array with N simple variables whenever it is possible. Another way of seeing it is to replace the array accesses for its respective values. This could enable some aggressive C-level optimizations. It is recommended that this optimization is done after function inlining, as described beforehand.

Given a Pallene module, we run the optimization in each function, by definition order. If the optimization can be safely applied, we can transform the code. The algorithm for a given function has 3 main steps:

(1) Firstly, for each array access, make sure that it can be transformed into a known value. We gather this information by abstractly interpreting the function with the abstract domain {constant , localvar , undefined, unknown}, with undefined meaning that no information could be determined (for example, whether the array was a parameter) and unknown meaning that the value could have multiple possible values and we can't determine which it would have at a said point. The localvar value indicates that the position value was assigned from a local variable. The abstract interpretation yields an array of states, one for each node of the command tree. Each state is formed by a mapping between variable names to their abstract values.
(2) Then, for each array access, make sure that the value hasn't escaped since the last known value. We also gather this information with abstract interpretation, using a simple boolean abstract domain, indicating that it has or hasn't possibly escaped. Similarly to the last step, it yields an array of states, each state containing a map of the variables to the boolean value corresponding whether it has escaped (defaults to false, hasn't escaped). We consider function calls and returns as escape points. We implemented this step alongside with the last one, so that the code is only interpreted once, but they are independent steps.
(3) For each array access, we check whether it has a known constant or local variable value and whether it hasn't possibly escaped. If both are true, we create the corresponding local variables and replace the array access.
(4) After all the replacing has been done, we check whether all of the array's accesses were transformed and whether the array hasn't possibly escaped (in this context, escaping is more concerned with other functions using it than modifying it). If both are true, we can eliminate the array creation and associated calls.

The main limitation in our implementation is that it only deals with arrays for now, but it should be simple to adapt it to also work with records. The escape analysis will stop any optimization if there is a function call between the last write and posterior reads. With a previous function inlining pass this limitation is mitigated. It only handles cases in which it has access to the array creation. It doesn't treat arrays that came as arguments or upvalues, but with a function inlining pass, this is also mitigated (at least with arrays created in Pallene).

Benchmark	Control run time avg. (sec)	Number of array accesses replaced	Difference in run time (%)
binarytrees	2.98	3	5.34
recordbinarytrees	4.28	0	0
binsearch	4.72	0	0
centroid	1.51	3	19.64
recordcentroid	2.20	0	0
conway	4.22	3	13.46
fannkuchredux	4.82	0	0
fasta	2.61	0	0
mandelbrot	7.77	0	0
matmul	9.67	1	15.10
nbody	3.22	21	12.46
recordnbody	7.97	0	0
objmandelbrot	9.24	5	55.41
recordobjmandelbrot	13.24	0	0
queen	2.09	0	0
sieve	28.94	0	13.51
spectralnorm	25.01	2	36.43

Table 2: Performance gains of performing the scalar replacement of aggregates optimization

We applied the optimization on programs with function inline already performed. The results discussed follow this and consider the function inline results as the control. We opted to benchmark this optimization on the result of the function inline because it enables many opportunities for optimizations. Also, this is the behavior that the compiler would execute.

This optimization has shown results in the range of 15% to 30% of reduction in total run-time, as we can see in Table 2.

It is interesting to note that the number of times that the optimization was performed did not directly correlate with the performance gains. This happens because the substitutions could be inside a loop or in frequently called functions. We can also see that the benchmarks that instantiate memory frequently for short uses (matmul, nbody and objmandelbrot) had the most impact, as expected from this optimization.

An interesting note of this optimization is how the results differed from the hand-optimized version to the implementation version. When we hand-optimized the code by modifying the emitted C, we would find results comparable to LuaJIT. For example, matmul would face a 55% reduction in run-time. When we implemented the algorithm, we found that in many cases the information that we thought that could be calculated in compile-time wasn't actually available, or at least our implementation couldn't handle such cases. An example of this is matmul, in which our optimizer only performed in 1 out of the 10 "possible" cases. LuaJIT can optimize these cases since it does its analysis in runtime, calculating the traces alongside the interpretation.

5.3 Renormalize

Lua implements its tables with two parts: an array part for integer indexes and a hash part for other key types. The array part can be internally resized if necessary. Since Pallene shares its data structures with Lua, Pallene must implement this mutation as well.

When accessing an array in Pallene, it is important to be sure that the accessed index is properly allocated. If it isn't, then the array is grown to fit it. Growing an array to a large size can be a quite costly operation, but more importantly, it can disable some C-level optimizations, since the C compiler can't guess when the growth will be performed.

We aim to reduce its occurrences by unifying, hoisting or deleting renormalize calls: We can unify two renormalize calls to the same array by only calling the renormalize to the larger index between the two; We can hoist a renormalize out of a loop if the accessed index is constant or is the loop variable (or the loop limit); We can delete a renormalize call if we know for sure that the array is already normalized, for example after its creation.

Of course, it isn't always safe to perform these optimizations. If there was a write or an escape (function call, in this context) between two renormalizes (or the same if it is inside a loop), we mark the case as unsafe and ignore it.

Before starting implementing the algorithm, it was necessary to decouple the renormalize operation from the array operations. Arrays had commands relative to creating them, setting their values and accessing their values. The renormalize operation was only emitted at the last step of the compilation process, whenever the array would be used. To the compiler, this operation was basically non-existent. In order to be able to modify the emission of renormalizes, we introduced a new command in the internal representation, NormalizeArr(arr , idx), and made the appropriate modifications when parsing the abstract syntax tree and when emitting the final C code.

Our algorithm then is straightforward: we collect the information for every NormalizeArr in the internal representation and apply the transformations if it matches one of the appropriate transformation cases.

(1) Firstly, for each NormalizeArr command, we calculate the possible range that the indexes (the second parameter) can have for each array (the first parameter). We do that with the use of an abstract interpretation using the abstract domain of integer ranges. While it is usual to represent each range as two integer values, the upper and the lower bound, we only need to find the maximum between values. For this reason, our abstract domain was the set of { unknown, undefined, localvar, upperbound}, with unknown meaning that a single upper bound couldn't be determined and undefined meaning that no information on the position could be retrieved. The localvar abstract value means that the upper bound for a given index is equivalent to the value of a local variable. This is particularly useful for hoisting, for example when the array is indexed with the for-loop variable. For each program point, the state is defined as a mapping between arrays to its possible indexes and a mapping between these indexes and their respective abstract values. The state also has information on the range of values of each local variable that is an integer.
(2) Then, also using abstract interpretation, we calculate which arrays have possibly escaped throughout the function body.

Possibilities of escape in this context are function calls (that could use the array as an upvalue) and returns.

(3) Finally, for each NormalizeArr we see if one of the following rules apply and transform it accordingly:
   (a) If the upper-bound of a NormalizeArr index can be determined, there was a previous NormalizeArr with a greater upper-bound and there wasn't any escape between them, it is safe to remove it.
   (b) If the upper-bound of a NormalizeArr index can be determined, there was a previous NormalizeArr with a smaller upper-bound and there wasn't any escape between them, it is safe to replace the latter for the former and deleting the smallest in the process.
   (c) If the upper-bound of a NormalizeArr index can be determined inside a loop (and the value didn't escape in any path of the loop), we can safely hoist it. In the case of the for loop, if the index can't be deemed constant, but its value is a variable, we can verify if it is the loop variable or the loop limit. In either cases we can safely hoist it replacing the index with the loop limit (even if it is another variable). This step works properly for nested fors.

As with scalar replacement of aggregates, we present the results over the benchmarks with function inline already applied.

We can see in Table 3 that we achieved results ranging from 5% to 23%. In particular, the benchmarks that heavily use arrays (and arrays of arrays), such as conway or nbody were particularly benefited from this optimization. Note the second and third column of the table. We counted first the unifications and then the hoistings. For example, if a for loop had 4 calls inside it and they all got unified into a single one and then hoisted, we would count 3 unifications (since it left one call) and 1 hoisting. At the same time, if there was a call inside nested loops and it was the case in which it was hoisted from both loops subsequently, we would only count as 2 hoistings, even though that the effect could be quadratic.

Benchmark	Control run time avg. (sec)	Number of calls unified	Number of calls hoisted	Difference in run time (%)
binarytrees	2.98	5	0	5.23
recordbinarytrees	4.28	1	0	0.91
binsearch	4.72	0	1	4.88
centroid	1.51	3	1	8.87
recordcentroid	2.20	0	0	0
conway	4.22	16	22	23.34
fannkuchredux	4.82	10	6	15.18
fasta	2.61	5	3	17.27
mandelbrot	7.77	0	0	0
matmul	9.67	3	6	8.22
nbody	3.22	60	6	20.19
recordnbody	7.97	0	3	6.94
objmandelbrot	9.24	10	0	7.4
recordobjmandelbrot	13.24	0	0	0
queen	2.09	0	3	6.9
sieve	28.94	0	5	8.98
spectralnorm	25.01	1	6	7.8

Table 3: Performance gains of performing the removal of renormalizes optimization

6 CONCLUSION

Pallene is a language created to be the system language counterpart for scripting using Lua as the application language. Pallene is a typed subset of Lua, making it convenient for the usual Lua programmer. The semantics of Pallene guarantees that every Pallene source code without its type annotations is semantically equivalent to the resulting Lua code. The Pallene compiler emits C that once compiled to a library can be required from Lua as any other C module.

Pallene can achieve better performance by emitting C code specialized for the type declarations. Pallene uses its type annotations to emit a C code with unboxed values, which can be more easily tracked by the C compiler in comparison with boxed values that dynamic languages commonly use. Pallene is a subset of Lua because it doesn't accept the some of the dynamic behaviors present in Lua, such as meta-programming features. By guaranteeing the absence of these behaviors, the Pallene compiler can emit C code with a simpler control-flow graph.

The resulting C has unboxed local variables and a simple control flow graph. These characteristics, associated with the knowledge of Lua's internal data structures allow the C compiler to generate efficient library modules. The access to Lua's internal data structures allows Pallene to interact safely with the Lua context without the need of using the C API, which usually introduces heavy overhead. The C compiler can apply its optimizations more comfortably with a source code that has a simpler control flow graph and values that can be tracked at the C level of abstraction. It would be significantly harder to do that with boxed values and the dynamic and unpredictable behavior that dynamic languages usually implement.

The Pallene compiler doesn't need to implement some optimizations because it can take for granted the optimizations already implemented by the C compiler. But, as we have discussed, it is not always the case that these optimizations can be performed in the C compiler. As Pallene works at a higher abstraction level than C, some of the semantics of Pallene isn't available for the C compiler to consider when optimizing the code. This lack of information can manifest in situations where an operation that has no side-effects for Pallene is seen by C as a function with unpredictable side effects.

In this work we argue that there are cases in which these optimizations that can't be performed by the C compiler can be implemented in the Pallene compiler. To support this argument, we found some opportunities of these optimizations, evaluate their possible performance impact by implementing them by hand and once validated, we implemented them in the Pallene compiler.

We found the possible optimizations by selecting several samples of Pallene code to examine the emitted C of each sample and their respective assemblies.

We described two of the opportunities of optimization: Replacing arrays for their scalar components, which can't be properly delegated to the C compiler and removing unnecessary calls that

Benchmark	Control run time avg. (sec)	Run time avg. with optimizations (sec)	Difference in run time (%)
binarytrees	3.63	3.49	3.90
recordbinarytrees	4.86	3.11	36.01
binsearch	4.82	4.63	3.86
centroid	1.51	1.15	23.53
recordcentroid	2.20	1.55	29.76
conway	4.80	2.43	49.41
fannkuchredux	4.82	2.72	43.55
fasta	2.87	2.43	15.19
mandelbrot	7.77	7.55	2.85
matmul	9.67	3.76	61.16
nbody	3.35	1.28	61.85
recordnbody	8.30	4.77	42.58
objmandelbrot	9.83	3.67	62.62
recordobjmandelbrot	14.09	10.25	27.28
queen	2.30	2.11	8.17
sieve	28.94	13.38	53.78
spectralnorm	27.19	12.14	55.37

Table 4: Performance gains of performing all the optimizations

exist only in C but are invisible to the Pallene user, such as array's renormalizations.

For each opportunity, we proposed a compiler change and demonstrated its possible gains by reproducing its effects by hand in the emitted C code. Once we checked the effectiveness of each transformation, we implemented the optimizations in the compiler.

We firstly implemented function inlining, since it could enable other optimizations at both Pallene and C level. Parameters doesn't have as much information as local variables, especially when talking about range analysis, one of the techniques that would be most used by the other optimizations. On the other hand, a function call can disable several optimizations; by writing over upvalues, it could change the caller's local state. Moreover, inlining function calls could give the C compiler more context to enable more of its optimizations. In fact, with only the inliner, we have seen a reduction of run time between 9% to 18% .

We then implemented scalar replacement of aggregates. It changes an array for its components, in the form of local variables. Implementing this optimization in the Pallene compiler cover the cases that the C compiler couldn't optimize. In the cases where the C compiler could fold aggregates it would still create, use and perform the associated runtime verifications, even without using its values, but with this optimization, these operations aren't emitted. Some of the benchmarks benefited from this optimization, with run time reductions varying between 13% to 55%.

The next optimization implemented was the reductions in renormalize calls. A certain property of the renormalize function allows us to eliminate some of the calls given the presence of others, based on its arguments. By collecting information on the range of the arguments, we could reduce the number of calls, reducing the run time of the benchmarks within 5% to 23%.

In order to evaluate all the optimization jointly, we applied the optimizations in the following order: Firstly we applied function inlining, since it would enable several cases for the other optimizations; Then we applied scalar replacement of aggregates. During SRA, we assumed that every array access is properly protected, since the renormalize reduction would remove unecessary guards to improve performance, it could disable some cases of scalar replacement; Then we applied the renormalize reduction optimization. We can see these results for all the optimizations applied in Table 4.

In this work we have argued that compilers that translate a higher-abstraction level language into C can't exempt themselves of implementing optimizations. While the C compiler can optimize several situations, the difference of abstraction level can impede some of the optimizations. Using the information available only at the level of Pallene, we implemented optimizations in the Pallene compiler that would be impossible to be performed by the C compiler. We have shown that the compiler modifications yielded significant results, which indicates that the principle of producing more specialized code based on the higher level context can generate useful optimizations.

REFERENCES

[1] Paul Biggar, Edsko de Vries, and David Gregg. 2009. A Practical Solution for Scripting Language Compilers. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii) *(SAC '09)*. Association for Computing Machinery, New York, NY, USA, 1916–1923. https://doi.org/10.1145/1529282.1529709

[2] Steve Carr and Ken Kennedy. 1994. Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience* 24, 1 (1994), 51–77.

[3] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

[4] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, 238–252.

[5] GCC. 2021. *Options That Control Optimization.* https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[6] Hugo Musso Gualandi. 2020. *The Pallene Programming Language.* Ph.D. Dissertation. Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

[7] Hugo Musso Gualandi and Roberto Ierusalimschy. 2018. Pallene: a statically typed companion language for lua. In *Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP 2018, Sao Carlos, Brazil, September 20-21, 2018*, Carlos Camarão and Martin Sulzmann (Eds.). ACM, 19–26. https://doi.org/10.1145/3264637.3264640

[8] Hugo Musso Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Science of Computer Programming* (2020), 102393.

[9] Roberto Ierusalimschy. 2008. Lua performance tips. *Last update: Wed Apr 27 09: 04: 45 BST 2016 (build 83)* (2008), 15.

[10] Martin Jambor. 2010. The new intraprocedural Scalar Replacement of Aggregates. *GCC Summit* (2010).

[11] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[13] Lukas Stadler, Thomas Würthinger, and H. Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *CGO '14*.

[14] Mark Wegman and Kenneth Zadeck. 1985. Constant Propagation with Conditional Branches. 291–299. https://doi.org/10.1145/318593.318659