Leonardo Krause Lipet Slipoi Kaplan

Optimizing the Pallene Compiler

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática, do Departamento de Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
April 2021

Leonardo Krause Lipet Slipoi Kaplan

Optimizing the Pallene Compiler

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Prof. Roberto Ierusalimschy
Advisor
Departamento de Informática – PUC-Rio

Profª. Noemi De La Roque Rodriguez
PUC-Rio

Prof. Francisco Figueiredo Goytacaz Sant'Anna
UERJ

Rio de Janeiro, April 23rd, 2021

Leonardo Krause Lipet Slipoi Kaplan

Graduated in computer science by PUC-Rio.

Bibliographic data

Kaplan, Leonardo Krause Lipet Slipoi

Optimizing the Pallene Compiler / Leonardo Krause Lipet Slipoi Kaplan; advisor: Roberto Ierusalimschy. – 2021.

72 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses. 2. Compiladores. 3. Linguagens dinâmicas. 4. Análise estática. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, to my brother and to my friends,
for their support and encouragement.

Acknowledgments

To my adviser Professor Roberto Ierusalimschy, for teaching me how to think pragmatically, how to write clearly and how to read critically. Also, for all the laughs and encouragements in the process of making this work.
To my colleagues of LabLua for all the discussions, questions and answers throughout these months of local and remote work. In particular, to Hugo Gualandi, for his patience and meticulousness when explaining simple and complex concepts and when reviewing my work, being it programmed or written.
And to the members of Campus Studio and the ones that made the initiative possible: Ivan Duffles, Professor Marcelo Gattass and Professor Sergio Bruni, for the trust placed in me.

To CNPq and PUC-Rio, for the aids granted, without which this work does not could have been accomplished.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Kaplan, Leonardo Krause Lipet Slipoi; Ierusalimschy, Roberto (Advisor). **Optimizing the Pallene Compiler**. Rio de Janeiro, 2021. 72p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Dynamic languages provide flexibility and simplicity in exchange for less compile-time information, leading to slower run times. Addressing this problem in the Lua context, the Pallene programming language appears as an alternative. In this work, we studied the current state of Pallene, searching for patterns that caused performance losses. Based on these patterns, we proposed and implemented several optimizations with the use of static analysis techniques.

Keywords

Compilers; Dynamic Languages; Static Analysis.

Resumo

Kaplan, Leonardo Krause Lipet Slipoi; Ierusalimschy, Roberto. **Otimizando o Compilador Pallene**. Rio de Janeiro, 2021. 72p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Linguagens dinâmicas provêm flexibilidade e simplicidade em troca de menos informação em tempo de compilação, o que resulta em perda de desempenho. Atacando este problema no contexto de Lua, a linguagem de programação Pallene surge como uma alternativa. Neste trabalho, examinamos o atual estado de Pallene, procurando por padrões responsáveis por perdas de desempenho. Baseado nestes padrões, propusemos e implementamos uma série de otimizações usando técnicas de análise estática.

Palavras-chave

Compiladores; Linguagens dinâmicas; Análise estática.

Table of contents

List of figures

List of tables

List of codes

A single coin
dropped in an empty flask
makes the most noise.

Ulla, *Talmud Bavli:*
Tractate Nezikin - Bava Metzia 85b.

1 Introduction

Dynamic languages provide flexibility and simplicity in exchange for less compile-time information, leading to slower run times.

There are several proposed paths to reduce the performance impact of dynamic languages: (IERUSALIMSCHY, 2008)(GUALANDI; IERUSALIMSCHY, 2020)

Rewriting the code in a faster, usually static typed, system language. Although, it may be too expensive to rewrite all of the code. Hence, the chosen strategy may revolve around scripting, rewriting just the performance-intensive modules, and operating via an interface to communicate between the two languages. Sometimes the overhead of these interfaces may offset the gains of the re-implementation.

Using a Just-in-time compiler. JITs require almost no modifications on the code and can significantly improve run-time performance. With the use of dynamic program analysis, it generates efficient code during the program execution, but its implementation is usually complex and not portable. While the use of JITs does not require any prior change in the code, they can be prone to optimization killers: code patterns that disable some optimizations and, even worse, can cause a significant slowdown in the code.

Using an optional type system. Optional types allow for the specification of static regions in the code base, enabling some of the compiler optimizations that the dynamic world would deem impossible. However, optional type systems usually just use the types for consistency, without using the type information to increase efficiency.

Addressing this problem in the Lua context, the Pallene programming language (GUALANDI; IERUSALIMSCHY, 2020) 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 despite its type system. Pallene's type system accommodates dynamically-typed Lua values, which would burden specialized emission. However, with the run-time checks, Pallene can determine whether the value is well-typed (in respect to its annotations) or whether it is a run-time error. Besides that, Pallene also forbids some of Lua's dynamic behaviors, such as monkey patching, polymorphic functions, and metamethods. These guarantees enable 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 C 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 be analyzing 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.

This document is structured as follows. In Chapter 2 we present the current state of the Pallene: what can or cannot be optimized in the C compiler and how Pallene is located amongst its alternatives (Lua, LuaJIT and C with the Lua API) in terms of performance. This chapter yields some results that will be used in Chapter 3, to propose some possible opportunities for optimizations. In Chapter 4 we describe the implementation of these optimizations and discuss their results. Finally, in Chapter 5 we present our conclusions.

2
The Current State of Pallene

We will be presenting an overview of the design of Pallene and its goals. One in particular is to be predictably efficient in comparison to its alternatives, namely PUC-Lua, LuaJIT and C using the Lua API (Lua-C API). Pallene can achieve competitive performance by compiling its code to C, making use of the optimizations already implemented by the C compiler. However, it isn't always the case that the optimizations can be performed.

We will be showing cases in which some optimizations can or can't be performed by the C compiler. In special, we will be discussing the case in which the Pallene compiler can be sure that a certain optimization could have been performed but the C compiler can't be sure. Also, we will be commenting some of the generated C code, in seek of opportunities of optimizations.

In the second section we will be comparing Pallene with its alternatives by comparing their run time. We will be discussing the cases in which Pallene didn't performed as well as the others. These cases could lead to optimizations that the alternatives implemented but aren't being performed in Pallene. If they aren't being performed in Pallene, it may be that the C compiler doesn't implement them or can't implement them. Either way, they could possibly be implemented or enabled by the Pallene compiler.

2.1
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 Codes 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 codes.

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 a greater performance. The system language usually achieve 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. We will be discussing this in depth later on.

```
function foo(N:integer):integer
    local a:integer = 10
    local b = 20
    return N + a + b
end
```

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

```
function foo(N)
    local a = 10
    local b = 20
    return N + a + b
end
```

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

2.2
Optimizations during C compilation

Because Pallene's design leaves most of the optimizations for the C 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.

2.2.1
Examining the assemblies

In this study, we used the GCC compiler, version 9.3.0. We used the `O2` optimization mode with the flags `-S -fverbose-asm -masm=intel -std=gnu11` to generate an assembly with commentaries. The `-S` flag indicates that we want to generate assembly, while `-fverbose-asm` asks for it to be commented. The `-masm=intel` flag emmits assembly using the intel (x86) syntax and `-std=gnu11` tells GCC to use the C11 standard with the GNU extensions.

We aggregated some Pallene code samples and examined the assembly of each. We compiled each sample to C using the Pallene compiler and and then to assembly using the C compiler. In Code 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
```

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

When we compile Code 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 (MUCHNICK, 1998) or constant propagation (WEGMAN; ZADECK, 1985), substitute variables with their values, when the values can be known in compile time. The C compiler can perform this optimization on normal C integers (GCC, 2021b). 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 Code 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 (JAMBOR, 2010) (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 Codes 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
```

Code 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
```

Code 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 Code 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.

2.2.2 The emitted C code

While having optimization killers is worrisome, there are still some possible opportunities for optimizations in the C code emitted by the Pallene compiler. Let's start by looking how the array is created and used in C when we compile the Code 5 into Code 6.

There is a lot of details in Code 6. Because of that, we have omitted some operations that do not concern our next commentaries with the ellipsis mark (...). The very first line is already confusing: While our Pallene code does not take any parameters, the C function takes 3 parameters. If there were a parameter in the Pallene function, it would appear as a fourth parameter in the C respective function. These first 3 parameters are internal to Pallene and necessary for the calls inside the function scope to be able to access Lua's runtime. The parameters are the `L` Lua state, which maintains the data structures; The `G` global context, pointing to userdata, globals and upvalues and the `base` stackvalue, which points to Lua's internal stack. For the following commentaries, just the first two will be relevant, but we will be talking about the three of them later on.

Note the variable declaration block between lines 4 and 8. This block shows more clearly what typed specialized emission means. Pallene's type integer becomes the `lua_Integer type`, which is just a conventional C integer, while the `Table` and `TString` types are some of Lua's internal structs. Pallene implementation could emit boxed values and just unbox them when needed, but instead it emits C values. In fact, if the function had an integer parameter, it would be unboxed by Pallene as soon as Lua called the function. The C equivalent would already receive the argument unboxed.

Now let's focus on the table initialization between lines 10 and 16. They are equivalent to the second line of the Pallene source. Firstly, in line 10, we create the table in Lua's runtime, with size 1. Then we set it as an heap value, on line 11. In line 12, we ask for a garbage collection pass if there is some debt. Then, between lines 13 and 16, we access the created `x1` array as a normal C array and set its value to the constant.

The block between lines 18 and 23 grabs the string operands from the upvalues in `G` and apply the concatenation in line 22. At this moment, the control is transferred to Lua. Omitting this last line re-enables the folding optimization.

Lines 12 and 24 have a similarity: they are invisible to the Pallene user. Line 24 normalizes the new array to size 1 (the third argument), it will grow the array allocated space to accommodate the requested size if necessary. Line 12 triggers a garbage collection pass if necessary.

The Pallene user doesn't know (and shouldn't know) that creating an array may trigger a collection pass or that accessing it may increase its internal size. However, these operations can lead to significant slowdowns. While the slowdowns can be mitigated with Pallene error handling techniques, removing these invisible calls when possible is always better. In this particular case, we can be sure at Pallene level that the array still have enough size to access the first element at line 24. We cannot remove the `condGC` call, that interacts with the garbage collector, at line 12. But if there was multiple `condGC` throughout the code, they could be reduced to only a few. With the appropriate adjusts, these calls to both `renormalize` and `condGC` could be reduced. We will discuss these invisible operations in more detail in the next chapter.

The next block, between lines 25 and 31, have two points of interest: the array access at line 26 and the error verification and handling between lines

27 and 29. Looking at the way the emitted C accesses arrays makes it clear why it could be folded, as lines 14 and 26 clearly references the same memory space. Also note how few validations exist: Besides the normalization call, it only checks whether the slot has the appropriate tag (at line 27), raising a runtime error if it isn't correct.

The way Pallene handle errors is quite efficient. Error treatment in Pallene is designed to be seen as an escape from the control flow graph. It should be clear to the C compiler that the control will not come back to the body if an error occurs so that it can optimize it accordingly. The way the compiler does this is with the `__builtin_unreachable()` macro at the end of the error handling function and declaring them with `__attribute__((noreturn))`. Either tells the C compiler that the specified block is an "escape arrow" in the control flow graph and will never affect the function's remaining. The function in line 28 use internally both of these annotations.

The `unreachable` macro is responsible for communicating to the C compiler that the control will never reach the said macro (GCC, 2021c). This can make the C compiler to generate a more linear code, usually placing the preceding block as a leaf of the control flow graph. It could also enable more optimizations that take advantage of knowing that the values can't change after a certain point.

The `noreturn` function attribute indicates that the function will never return (GCC, 2021a). It should have roughly the same effect of adding an unreachable macro after every call of the function.

In line 27, there is the `PALLENE_UNLIKELY` macro, which uses the `__builtin_expect(!!(x),0)` macro internally. This second macro indicates the expected value of the conditional, for a percentage of the cases (which can be configured but Pallene uses the default, 90%). It can enable branch prediction optimizations (GCC, 2021c). In particular, branches marked as unexpected will appear later in the assembly. This repositioning makes better use of the already fetched instructions since a long jump would discard them.

2.2.3 Calling Pallene from Lua

When Lua calls a Pallene function, it first calls the function's associated entry point function, which will perform some run-time checks, unbox the arguments and then properly call the C function, returning the value to Lua appropriately. When a Pallene function is called from Pallene, it doesn't need to call the entry point function.

```
 1 static int foo_lua(lua_State *L)
 2 {
 3     StackValue *base = L->ci->func;
 4     CClosure *func = clCvalue(s2v(base));
 5     Udata *G = uvalue(func->upvalue[0]);
 6
 7     <arity checks>
 8
 9     lua_Integer ret1;
10     ret1 = foo(L, G, L->top);
11
12     <passing the returned value (ret1) to Lua>
13     return 1;
14 }
```

Code 7: Example of an entry point function in C

As we have seen in Code 6 in the last section, Pallene-emitted functions have 3 parameters. The corresponding arguments are calculated independently of their actual usage. This could be a source of unnecessary slowdowns, specially on functions that are called inside a loop in Lua.

To demonstrate these operations more concretely, let's look at Code 7, an example of the entry point of `foo`, a simple function that takes no arguments and returns a constant integer. The entry point function name follows the convention of being the Pallene's function name with the `_lua` suffix. At lines 3, 4 and 5 we see the values being calculated and in line 10 we see them being passed to the `foo` function. The function doesn't use any of them internally. To obtain the `base` value, it is necessary to calculate 3 indirections. By expanding the macros on the right hand side of `func` and `G`, we discover that we calculate 4 and 6 indirections each, respectively. These indirections can add up to significant overhead if the function is called frequently from Lua.

2.2.4 Data structures

One of Lua's main features is the table data structure, its only data structure. It is an associative array that can be indexed with and store any of the language values with the exception of `nil`. For the Lua programmer, a table have no fixed size, growing automatically and invisibly when needed, independently of the index's type. However, the table is implemented quite differently for integer keys than it is for other types of key.

Tables are implemented with two parts: an array part for integer keys and a hash part for the other types of key. We have already seen, in Subsection 2.2.2, the array part being used in Pallene and in its compiled counterpart. Besides integers keys, Pallene only supports string keys. After declaring the table type structure, it can be used similarly as in Lua.

We can see how we declare a table with string fields in Code 8. In this function, we receive two float values and return a table with fields `x` and `y` containing the two values. Note the table declaration structure in lines 1 and 2. Since we are returning the value declared in line 2 in line 3, the type declarations must match. Also note in line 2 how the string fields are used in the table initialization.

```
1 function new(a : float, b : float): {x: float, y: float}
2     local v : {x: float, y: float} = {x = a, y = b}
3     return v
4 end
```

Code 8: Example of tables with string fields in Pallene

It may be convenient to write the type declaration just once, in order to ease refactorings and maintain uniformity. Pallene supports the `typealias` syntax, as seen on the first line of Code 9. It is then used on lines 3 and 4, replacing the repeated type declaration.

It is interesting to note that Pallene uses structured typing, in opposition to nominal typing, meaning that two records with the same structure but different aliases are compatible for all operations. Because of that, we do not need to consider type-aliases when talking about records.

```
1 typealias point = {x: float, y: float}
2
3 function new(a : float, b: float): point
4     local v : point = {x = a, y = b}
5     return v
6 end
```

Code 9: Example of the typealias syntatic sugar

Pallene's performance when using strings or integers keys should be proportional to their respective costs in Lua. However, we can verify that using string keys introduces significant overhead in comparison to the usage of integers keys. That is, if using an integer key takes half the time that it takes to use a string key in Lua, this should be similar in Pallene. We will be investigating this overhead in the next chapter.

2.2.5 Conclusion

In this section we have seen that interleaving transference of control to Lua with access to heap values hinders constant folding of heap values. We can solve this transforming heap values into stack values when appropriate. We have briefly discussed how some commands at Pallene level may introduce other commands at the C level during emission, such as garbage collection checks and array renormalization. These commands can introduce significant overhead and can too be optimized. We then discussed how errors are handled efficiently and what happens when Lua calls a Pallene function, with possible points to be optimized. Lastly, we briefly described how tables are implemented and noticed that the performance in using different key types in Pallene isn't proportional to Lua's , which may indicate an optimization opportunity.

2.3 Comparing with the alternatives

Pallene alternatives are Lua, Lua-C API, or LuaJIT. One form of understanding how Pallene could be more efficient is to compare its performance to its alternatives in a set of benchmarks. We may find opportunities of optimizations in the cases where Pallene took more time than its alternatives. We will present the programs used to evaluate the

2.3.1 Benchmarks

To measure the impact of each optimization, we made a set of benchmarks based on the ones used in other works involving Pallene (GUALANDI, 2020)(GUALANDI; IERUSALIMSCHY, 2020)(GUALANDI; IERUSALIMSCHY, 2018). We will be describing 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. 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. Each benchmark was run 10 times using the perf program. The results are summarized in Figures 2.1 and 2.2. Figure 2.1 lists the average time in seconds for each benchmark and alternative. Figure 2.2 presents the results of just Pallene and LuaJIT in comparison with Lua. The results are presented with average time for each benchmark divided by the average time for the reference Lua interpreter.

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. **binsearch:** it performs a simple binary search in a sequential array with the size passed as a parameter.

3. **centroid:** it creates N points and finds the centroid between them.

4. **conway:** simulates the Conway's game of life in a 40 by 80 grid, for a given number of steps.

5. **fannkuch-redux:** given a positive integer N, it builds a sequential array of 1 to N. Then, for each of the $N!$ permutions, it does a procedure in which the elements of the vector are swapped in position until the first position contains the number 1.

6. **fasta:** it calculates the similarity between strings representing DNA sequences.

7. **mandelbrot:** it generates the mandelbrot set of a given integer, N, outputing an image in the netpbm format.

8. **objmandelbrot:** it is similar to the mandelbrot benchmark but uses arrays instead of pairs of integers.

9. **matmul:** it multiplies a $N \times N$ matrix with itself a number of times.

10. **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.

11. **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.

12. **sieve:** it computes the sieve of Eratosthenes for a given positive integer N, calculating all the prime numbers up to N.

13. **spectralnorm:** it calculates the spectral norm of a $N \times N$ square matrix for a given N.

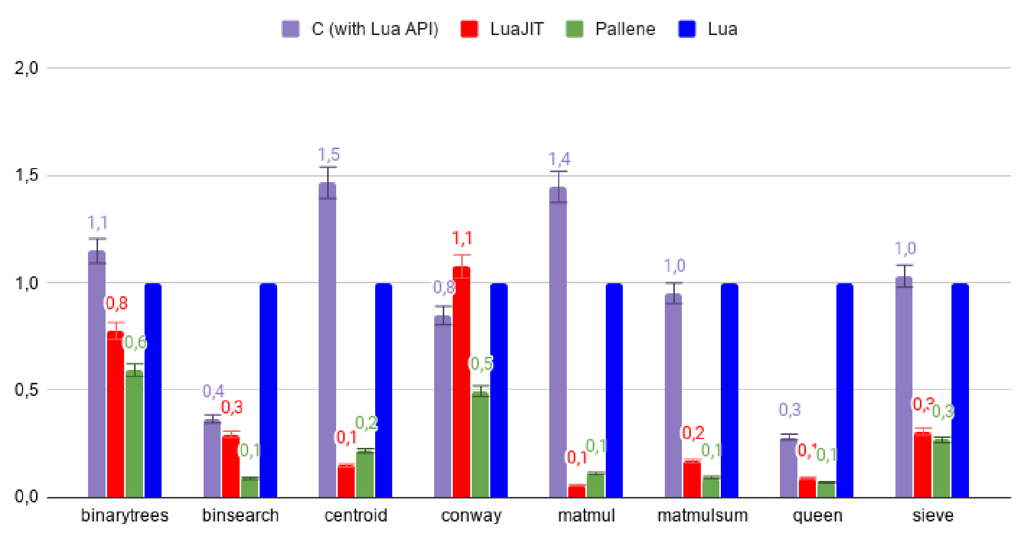

Figure 2.1: Time spent on each benchmark normalized to Lua. A higher bar means more time spent.

In this set of benchmarks, we see that Pallene is almost always the best alternative, except for the centroid and matmul cases, in which the LuaJIT implementation is the best. We will discuss these two later on in this section.

We decided to investigate further the relationship between Pallene and LuaJIT as speed-up methods of Lua. We present a more diverse set of benchmarks where we could pinpoint other cases in which LuaJIT is more performant than Pallene.

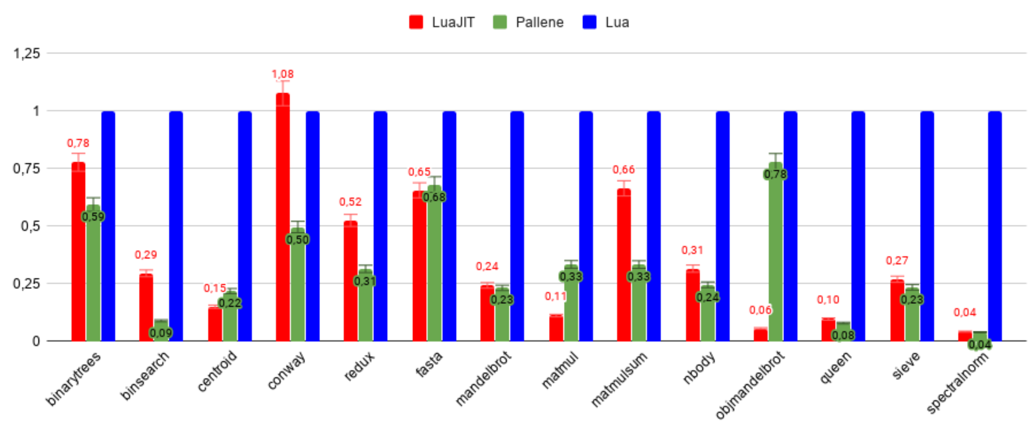

Figure 2.2: Time spent on each of the more diverse benchmark set, normalized to Lua. A higher bar means more time spent.

In these benchmarks, we can see that Pallene is predominantly faster or similar to LuaJIT, except for 3 cases in which LuaJIT is significantly faster (centroid, matmul, and objmandelbrot). These cases are objects of interest for us since they can have similarities that could lead to optimizations opportunities.

Matmul is a particular case that was previously analyzed in (GUALANDI; IERUSALIMSCHY, 2020): LuaJIT implements NaN-boxing, which packs both a Lua value and its type information in a single 8 byte floating number, in opposition to the 16 bytes used by both Lua and Pallene. This difference reduces cache misses in LuaJIT, increasing its performance.

While centroid and objmandelbrot can also be affected by NaN-boxing, a closer inspection of these benchmarks' code in comparison to the others shows that they perform more table inline declarations that are quickly used and then collected as garbage.

LuaJIT performs an optimization (the combination of allocation sinking and store sinking to perform scalar replacement of aggregates) to deal with this exact issue (PALL, 2012). The optimization consists of removing temporary allocations by transforming a table into several local variables equivalent to it. We believe that implementing a similar optimization will lead to similar gains.

```c
static lua_Integer f(lua_State *L, Udata *G, StackValue *base
    ) {
    ...

    Table *x1; /* x */
    lua_Integer x2; /* y */
    TString *x3; /* s */
    lua_Integer x4; /* z */
    lua_Integer x5;

    x1 = pallene_createtable(L, 1, 0);
    sethvalue(L, ..., x1);
    luaC_condGC(...);
    {
        TValue *slot = &x1->array[1 - 1];
        setivalue(slot, 43);
    }
    x2 = 71;
    {
        TString *ss[2];
        ss[0] = tsvalue(&G->uv[0].uv);
        ss[1] = tsvalue(&G->uv[1].uv); ;
        x3 = pallene_string_concatN(L, 2, ss);
    }
    pallene_renormalize_array(L, x1, 1, ...);
    {
        TValue *slot = &x1->array[1 - 1];
        if (PALLENE_UNLIKELY(!ttisinteger(slot))) {
            pallene_runtime_tag_check_error(L,...);
        }
        x4 = ivalue(slot);
    }
    x5 = intop(+, x4, x2);
    return x5;
}
```

Code 6: Fragment of the sum of heap constants with string concatenation example compiled to C

3
Opportunities for optimizations

In the last chapter we have seen some situations in the emitted C that could be associated with slowdowns. In this chapter, we will be proposing changes to the compiler to address these situations. For each proposed change, we will be presenting their expected impact, by performing each optimization by hand on the C code. We will show examples of these transformations by hand as well.

3.1
Scalar replacement of aggregates

In the last chapter we have seen two cases in which Pallene could be more efficient in relation to its alternatives. For these particular cases, LuaJIT can be faster by implementing scalar replacement of aggregates, which the C compiler also implements but doesn't always perform, as we have also discussed in Chapter 2.

The first issue previously seen is the case in which the C compiler can't perform constant folding over arrays' values. Transferring the control to Lua between accesses disables the optimization. However, even when the optimization can be performed by the C compiler, the result is folded but the array is still created, introducing overhead. By implementing the scalar replacement optimization at Pallene level, we can prevent the array of being created, replacing it by scalar values when possible.

We reproduced the effects of this optimization on a set of benchmarks, reducing the total run time to a less than a tenth of the original time. The sample would create an array of integers and return its first value. We replaced it into scalars corresponding to its initial values, eliminating the overhead of creating and accessing it.

We can demonstrate an example of scalar replacement done by hand by compiling the Code 5 (described in Subsection 2.2.1) to C and transforming it. We present a fragment of the compiled code below, in Code 10. As before, the parts that aren't relevant to our comments are omitted with the ellipsis mark (...). In line 3, we modified the declaration of `x1` to be an integer. Then, in lines 8 to 10, we deleted the array creation process. Next, in lines 13 and 14, we substituted the array slot acquisition and the write on it for a simple integer assignment. Finally, when reading the value of the array between lines

```
1  static lua_Integer f() {
2      ...
3      Table *x1;  /* x */
4      lua_Integer x2; /* y */
5      lua_Integer x3; /* z */
6      lua_Integer x4;
7
8      x1 = pallene_createtable(L, 1, 0);
9      sethvalue(L, s2v(base + 0), x1);
10     luaC_condGC(L, L->top = base + 1, (void)0);
11     {
12         TValue *slot = x1->array[1 - 1];
13         setivalue(slot, 43);
14     }
15     x2 = 71;
16
17     <string concatenation>
18
19     pallene_renormalize_array(L, x1, 1, ...);
20     {
21         TValue *slot = x1->array[1 - 1];
22         if (PALLENE_UNLIKELY(!ttisinteger(slot))) {
23             pallene_runtime_tag_check_error(L,...);
24         }
25         x3 = ivalue(slot);
26     }
27     x4 = intop(+, x3, x2);
28     return x4;
29 }
```

Code 10: Fragment of hindered constant folding example before the scalar replacement transformation by hand

20 and 26, we removed the checks in lines 20, 23 and 24 and changed the array access on lines 22 and 26 for a simple integer-to-integer assignment.

The result of the process described in the last paragraph can be seen on Code 11. We can see that this code is much shorter and simpler. Line 3 shows the new integer type for `x1`, while lines 8 and 14 show its being used directly, without the whole array creation, manipulation and verification overhead.

To perform the optimization, we need to be sure of the value of an accessed position of the array at that said point. When the array is created in the same scope, this is usually easy to do. When the array is a parameter of the function or an upvalue, this becomes more difficult. Because of this, we decided to inline some of these calls that received arrays, in order to have more information on the arrays that we were trying to replace. As we continued the process of hand optimizing the C code, we noted that the function-inline

```c
static lua_Integer f() {
    ...
    lua_Integer x1; /* x */
    lua_Integer x2; /* y */
    lua_Integer x3; /* z */
    lua_Integer x4;

    x1 = 43;
    x2 = 71;

    <string concatenation>

    {
        x3 = x1;
    }
    x4 = intop(+, x3, x2);
    return x4;
}
```

Code 11: Fragment of hindered constant folding example after the scalar replacement transformation by hand

optimization could be an useful optimization to be implemented.

3.2 Function inlining

We can demonstrate the case of function inlining enabling optimizations at Pallene level with Code 12. This program simply returns 20 when `foo` is called. It creates an array with 20 in its first position and calls `boo` passing the array, which returns the first position. The return of `boo` is then returned. This is a short example of how function calls could hinder scalar replacement of aggregates. If the array access in line 2 could see the array creation on line 6, it would be trivial to replace it with a scalar. When we inline the call to `boo` in `foo`, we have enough information to perform the optimization on `foo`. It is important to keep `boo` as it is, since it could be called from Lua or from other contexts.

We can show the process of inlining the function in C by compiling Code 12 into Codes 13 and 14. Both of these compiled codes were emitted in the same C file, but for a better presentation we separated them. In Code 13 we can see the C equivalent of the `boo` function, while in Code 14 we can see the equivalent for the `foo` function. As usual, irrelevant fragments were ommited with the elipsis mark (...).

On Code 14, we want to replace the function call in line 15 by the function body of Code 13. We also want to eliminate the `restorestack` command in line 16, because it corrects the Lua stack after a function call (that won't

```
function boo(x : {integer}) : integer
    return x[1]
end

function foo(): integer
    local x : {integer} = {20}
    local y = boo(x)
    return y
end
```

Code 12: Example of a situation in which scalar replacement is hindered by function calls

be present anymore). This transformation results in Code 15. To perform the transformation, we executed three main steps.

We firstly created local variables equivalent to the argument passed to `boo`. The argument `x1` in line 15 of `foo` and the parameter of line 2 of `boo` became the declaration of line 20 in Code 15. In line 25 we used the new name accordingly.

We then copied `boo`'s body to the body of `foo`, in lines 21 to 31 of Code 15. The first copied line, in line 21, is analogous to the `restorestack` command that we mentioned before. It is only used when a function is called, and so, can be removed (its removal is represented by the line being commented). It is important to adjust the internal local variables of the inlined function to have non-conflicting names with the variables already present in the body of the caller function. In line 22, we renamed the variable `x2` of `boo` into `localvar1`, then, in line 29 and 31, we used it.

Lastly, it is important to create local variables corresponding to the returned values. In line 31, we created the `ret1` variable and assigned the returned value to it. This corresponds to line 16 of Code 13. We then replaced the call for the returned value.

Because no calls were performed, we can safely comment line 34, as we discussed before.

3.3 Invisible calls

In the last chapter, we have discussed internal calls that doesn't appear in a Pallene source but are emitted to C, such as the `renormalize` and the `condGC` functions. These could be prevented of being emitted knowing the details of their effects.

```c
static lua_Integer boo(..., Table *x1  /* x */) {
    ptrdiff_t base_offset = savestack(L, base);

    lua_Integer x2;

    pallene_renormalize_array(L, x1, 1, PALLENE_SOURCE_FILE,
       7);
    {
        TValue *slot = &x1->array[1 - 1];
        if (PALLENE_UNLIKELY(!ttisinteger(slot))) {
            pallene_runtime_tag_check_error(L,...);
        }
        x2 = ivalue(slot);
    }
    return x2;
}
```

Code 13: The `boo` function of Code 12 compiled to C

```c
static lua_Integer foo(...) {
    luaD_checkstack(L, 1);
    ptrdiff_t base_offset = savestack(L, base);

    Table *x1; /* x */
    lua_Integer x2; /* y */

    x1 = pallene_createtable(L, 1, 0);
    sethvalue(L, s2v(base + 0), x1);
    luaC_condGC(L, L->top = base + 1, (void)0);
    {
        TValue *slot = &x1->array[1 - 1];
        setivalue(slot, 20);
    }
    x2 = boo(L, G, base + 1, x1);
    base = restorestack(L, base_offset);
    return x2;
}
```

Code 14: The `foo` function of Code 12 compiled to Cd

```
static lua_Integer foo(
    lua_State *L,
    Udata *G,
    StackValue *base
) {
    luaD_checkstack(L, 1);
    ptrdiff_t base_offset = savestack(L, base);

    Table *x1; /* x */
    lua_Integer x2; /* y */

    x1 = pallene_createtable(L, 1, 0);
    sethvalue(L, s2v(base + 0), x1);
    luaC_condGC(L, L->top = base + 1, (void)0);
    {
        TValue *slot = &x1->array[1 - 1];
        setivalue(slot, 17);
    }

    Table *arg1 = x1;
    //ptrdiff_t base_offset = savestack(L, base);
    lua_Integer localvar1; // formely x2
    pallene_renormalize_array(L, arg1, 1, PALLENE_SOURCE_FILE
        , 7);
    {
        TValue *slot = arg1->array[1 - 1];
        if (PALLENE_UNLIKELY(!ttisinteger(slot))) {
            pallene_runtime_tag_check_error(L,...);
        }
        localvar1 = ivalue(slot);
    }
    lua_Integer ret1 = localvar1;

    x2 = ret1;
    //base = restorestack(L, base_offset);
    return x2;
}
```

Code 15: The `foo` function with the body of `boo` inlined

3.3.1 The `condGC` call

`condGC` is a function that checks whether Lua's garbage collector has any debt and calls for a collection pass, if necessary. A call to `condGC` and function calls are the only points of potential garbage collection in Pallene. To prevent any kind of memory leaks, Pallene emits a `condGC` call after every assignment to a variable with garbage-collectable type (strings, functions, data structures and anys). However, these calls could be reduced.

The `condGC` call only warns Lua's garbage collector that memory was allocated, but not how much memory was allocated. Because of that, we can call it once after all the allocations have happened. One possible heuristic is to emit this call only at the end of the function. We cannot follow this heuristic, because there is always the possibility of the presence of an infinite loop with allocations inside. These allocations could potentially consume infinite memory before the first `condGC` call. Because of that, we resorted in performing a similar heuristic: We remove all `condGC` calls and add one before each control-flow branching. This is the same as having one call for each basic block. Note that loop iterations are branches of control, so we cannot hoist the call to before or after the loop.

A short example of the `condGC` reduction can be seen on Codes 16 and 17. These are just fragments of the whole program, with all the irrelevant parts omitted with the ellipsis mark. In Code 16 we create a number of tables. We firstly create two, in lines 2 and 6. As we can see in lines 4 and 8, each table creation have a respective `condGC` call afterwards. Then, in line 12, inside of a potentially infinite loop, we create a new table at each iteration, accompanied by its `condGC` on line 14.

We can also note the `sethvalue` calls, they are responsible for registering the newly created object in the Lua stack (effectively making it accessible to Lua). Pallene already perform an optimization of just emitting calls to this function on values that can still be alive after the end of the function body. If the values will surely die in the local scope, they don't need to be passed to Lua nor need to be tracked by Lua's garbage collector.

In Code 17 we can see the result of reducing `condGC` calls. As we have discussed in this section, our heuristic is to remove all `condGC` calls inside of a basic block and insert one at the end, if there was at least one `condGC` call in the block. We did this for the first of the two basic blocks present in the fragment shown in Code 17. The first basic block is between lines 2 and 8, while the second one is between lines 10 and 16. In the second block we couldn't remove the call, as discussed before.

```
...
x5 = pallene_createtable(...);
sethvalue(...);
luaC_condGC(...);

x6 = pallene_createtable(...);
sethvalue(...);
luaC_condGC(...);

while (1) {
    if(...) break;
    x7 = pallene_createtable(...);
    sethvalue(...);
    luaC_condGC(...);
    ...
}
...
```

Code 16: Fragment of a Pallene code compiled to C, before the condGC elimination optimization was applied by hand

On performing the transformation by hand we could analyze the code and be sure that the loop would surely terminate, hoisting the `condGC` to after the loop. However, this analysis is quite hard to be programmed, even undecidable for most languages. To experiment only with the effects of what can be implemented, we opted to omit the rest of the loop and its breaking condition, assuming all loops as potentially infinite.

We could see reductions of up to 10% of total run time when applying these transformations in Pallene functions with a handful of memory allocations. The functions were called repeatedly inside a tight loop in Lua.

3.3.2 The `renormalize` function

The Pallene compiler adds a call to `pallene_renormalize_array` (`renormalize` for short) before every array access. In several situations, these calls could be omitted or could be hoisted outside of a loop.

We can perform an optimization using an interesting property of the function: After checking whether a table has a certain size, one can safely assume that it has at least that size. In other words, one can safely remove any `renormalize` that checks for a smaller size since the last `renormalize` call to the same table (given that the table hasn't been possibly enlarged).

By performing some range analysis, we can substitute several calls for a single one that checks only the upper bound. If the array was previously

```
        ...
        x5 = pallene_createtable(...);
        sethvalue(...);
        //luaC_condGC(...); // removed

        x6 = pallene_createtable(...);
        sethvalue(...);
        luaC_condGC(...);

        while (1) {
            if(...) break;
            x7 = pallene_createtable(...);
            sethvalue(...);
            luaC_condGC(...); // cannot be removed
            ...
        }
        ...
```

Code 17: Fragment of a Pallene code compiled to C, after the condGC elimination optimization was applied by hand

created with sufficient size, we can even remove this remainder call. Removing a `renormalize` call from a program without loops had an impact of 5%, while hoisting it had an impact of 15%. These programs were called repeatedly from Lua in a tight loop.

We can see examples of how we applied these transformation by hand with a fragment of the centroid benchmark compiled to C, shown in Codes 18 and 19, representing the fragments before and after the transformation, respectively. This code is an iteration of `i` from one to N, accessing the `i`-th position of the `A` array and the first two positions of the `B` array.

In lines 4, 10 and 17 of Code 18, we can see calls for `renormalize`. `renormalize`'s arguments are, in order, a reference to the Lua state, the array to be checked and the size it should have. Also, it receives debug information, which is omitted here with the ellipsis mark. Note that the first call checks for the `i`-th position of the `A` array, with `i` being the loop variable. The other two calls check for the first and second positions of the `B` array, respectively.

The first transformation that we can apply is to hoist the first call. We can do that by extracting the call to outside the loop, in special before the loop, and adjusting the checked position (the third argument) to its upper bound. We can see this on the first line of Code 19. We know that `i` can be any integer value up to N, so its upper bound is N. This can be known in compile time for some cases of loops. In particular, the presented case, in which the initial value, the limit value and the step are known, is trivial to know the loop variable's range and subsequently, its upper bound. In the next chapter, we will be talking about how to know these ranges in a more general way.

The second possible transformation is in fact simpler than the first. In lines 10, we firstly check for the first position of `B` and in line 17, we check for `B`'s second position. As we have discussed, we can unify this calls into a single call checking for the second position. This new call should replace the first call, as we can can see on line 11 of Code 19.

We can generalize these transformations in three main cases in which we could improve the code by working around `renormalize` calls:

The simplest case is on inline table declaration. Pallene creates a table with a given number of slots. However, subsequent `renormalize` calls that checks for indexes smaller than the initialized size are still emitted. We could maintain the size information of the created array and prevent the emission of these calls.

The next case is on unifying several `renormalize` calls without the table being modified between those calls. Since the table wasn't modified, the calls would have the same effect and could be unified into the first. We just have to make sure of adjusting the checked index for its maximum value between the unified calls.

The last case, but the most impacting, is hoisting `renormalize` calls by analyzing whether the size checked is constant (always accessing the same position during the loop). To numeric loops, we could implement an heuristic on whether the size checked is the same variable as the iteration variable of the loop. If the `renormalize` call fulfils this requirement, then it can be hoisted by substituting its size checked with the upper bound of the iteration variable (the limit of the loop). A similar approach can be done for decreasing numerical loops (but substituting the variable by the initial value).

3.4
Calling Pallene from Lua

We have discussed in the last chapter that Pallene creates an entry point function for each of its functions. This entry point function is exposed to Lua and bridges the transference of control, arguments and returns between both sides. We have also seen that all Pallene functions, when compiled to C, have at least three internal parameters, independently of the actual number of parameters defined at the Pallene level.

The entry point function calculates the arguments for the internal parameters of a Pallene function independently of their actual use. We experimented removing these calculations when unnecessary. For example, in line 12 of Code 20, we can see that the `G` variable is passed to the `foo` function. This function

```
for(int i = 1; i <= N; i++)
{
    {
        pallene_renormalize_array(L, A, i, ...);
        TValue *slot = &A->array[i - 1];
        < slot run-time type check and error handling >
        x8 = hvalue(slot);
    }
    {
        pallene_renormalize_array(L, B, 1, ...);
        TValue *slot = &B->array[1 - 1];
        < slot run-time type check and error handling >
        x9 = fltvalue(slot);
    }
    x3 = x3 + x9;
    {
        pallene_renormalize_array(L, B, 2, ...);
        TValue *slot = &B->array[2 - 1];
        < slot run-time type check and error handling >
        x10 = fltvalue(slot);
    }
    x4 = x4 + x10;
}
```

Code 18: Fragment of the centroid benchmark compiled to C, before the renormalize elimination optimization applied by hand

```
pallene_renormalize_array(L, A, N, ...);
for(int i = 1; i <= N; i++)
{
    {
        TValue *slot = &A->array[i - 1];
        < slot run-time type check and error handling >
        x8 = hvalue(slot);
    }
    pallene_renormalize_array(L, B, 2, ...);
    {
        TValue *slot = &B->array[1 - 1];
        < slot run-time type check and error handling >
        x9 = fltvalue(slot);
    }
    x3 = x3 + x9;
    {
        TValue *slot = &B->array[2 - 1];
        < slot run-time type check and error handling >
        x10 = fltvalue(slot);
    }
    x4 = x4 + x10;
}
```

Code 19: Fragment of the centroid benchmark compiled to C, after the renormalize elimination optimization applied by hand

```
1  static int foo_lua(lua_State *L)
2  {
3      StackValue *base = L->ci->func;
4      CClosure *func = clCvalue(s2v(base));
5      Udata *G = uvalue(func->upvalue[0]);
6
7      int nargs = lua_gettop(L);
8      if (PALLENE_UNLIKELY(nargs != 0)) {
9          pallene_runtime_arity_error(...);
10     }
11
12     foo(L, G, L->top);
13     return 0;
14 }
```

Code 20: Example of an entry point function for a function foo that takes no arguments and have no returns

does not use `G` internally. In fact, the `G` variable is just used if an upvalue is requested. The only other possible use is to pass it as an argument to other functions (as an internal parameter). Removing the calculations for `G`, in lines 3,4 and 5, and substituting the value of G for `NULL` showed reduction of approximately 5% of the runtime when calling the Pallene function from Lua in a tight loop.

3.5 Records

We discussed how differences in the types of the key can introduce overhead when using tables in Pallene. In particular, we have shown the contrast between arrays and records. To further investigate this question and to provide more detailed data, we implemented benchmarks using records. It was necessary to introduce new programs to our benchmark suite (described in Section 2.3.1 because it had no representative on the use of records.

We introduced four new programs to our benchmarking suite. They were created from the binarytrees, centroid, nbody and objmandelbrot programs. These programs used arrays with fixed sizes and meaningful positions, for example implementing a two dimensional point as an array of two positions. They seemed appropriated to be rewritten as records.

We will be exemplifying the adaptation of these benchmarks with a fragment of the objmandelbrot benchmark. In the fragments presented in Codes 21 and 22, there are three functions: `new`, that takes two float parameters and returns a two-dimensional point (represented as an array of floats or a record, depending on the implementation); `clone`, that takes a two-dimensional

```
 1 function new(x: float, y: float): {float}
 2     return {x, y}
 3 end
 4
 5 function clone(x: {float}): {float}
 6     return new( x[1], x[2] )
 7 end
 8
 9 function conj(x: {float}): {float}
10     return new( x[1], -x[2] )
11 end
```

Code 21: Fragment of the objmandlbrot benchmark in Pallene

point and returns a newly allocated copy of it; And `conj`, that takes a two-dimensional point and returns its newly-allocated conjugate.

The procedure that we performed to write these alternatives was straightforward. For example, in the objmandelbrot benchmark, which is partially shown in Code 21, we substituted the array-of-floats return type of each function for the newly declared `point` type, declared in the first line of Code 22. We can visualize this transformation by comparing the return types of lines 1, 5 and 9 of Code 21 with their counterparts (lines 2, 8 and 12 of Code 22).

Other modifications include the parameters of functions `clone` and `conj` and their respective uses, changing the integer indexing (lines 6 and 10 of Code 21) with string indexing (lines 9 and 13 of Code 22). The body of the `new` function was the one that changed the most. In Code 22, a new local variable was introduced, `v`, with their structure declared and used. All arrays of the same type have the same structural type, so this introduction wasn't necessary in Code 21.

We then executed the new benchmarks in the same form as described in Section 2.3.1. We could observe that the overhead was of approximately 30%, as we can see on Table 3.1. We then investigated the sources of the overhead by commenting parts of the emitted code and seeing the impact of each modification. We have verified that wasn't the actual usage of the records, but their creation that was responsible for the most significant part of the slowdown. In special, creating the string keys was quite costly. We will describe the results of this investigation and some solutions implemented in the next chapter.

```
typealias point = {x: float, y: float}

function new(x : float, y: float): point
    local v : point = {x = x, y = y}
    return v
end

function clone(p : point) : point
    return new( p.x, p.y )
end

function conj(p : point) : point
    return new( p.x, -p.y )
end
```

Code 22: Fragment of the record-objmandlbrot benchmark in Pallene

Benchmark name	Run time using arrays (sec)	Run time using records (sec)	Difference in run time(%)
record-binarytrees	1.66	2.2	32.53
record-centroid	1.8	2.4	33.33
record-objmandelbrot	1.28	1.72	34.38
record-nbody	1.56	2.06	32.05

Table 3.1: Overhead introduced by the use of records in Pallene in comparison to arrays

4 Implementation and evaluation of the optimizations

In the last chapter we have described the optimizations that we planned to implement. In this chapter, we will describe their implementations and results.

The array renormalization and the SRA optimizations requirecontext information to be performed. Both need escape analysis,while SRA also needs some alias analysis to track the aggregates' fields and renormalize needs range analysis to determine the upperbounds for each array size-check. We opted to gather this informationwith the use of abstract interpretation.

Firstly, present a short overview of abstract interpretation, the technique used by most of our implementations to obtain the required information at Pallene level to perform the optimizations. After that, we will give a brief description of each program used to measure the run time for each optimization. Finally, we will be discussing each optimization implemented.

4.1 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 (RICE, 1953). We can illustrate how interpreting programs concretely is not computable by simply trying to find the final value of `x` in Code 23. The program has an infinite loop, so the compiler would be stuck in this interpretation if it tried to do it concretely. With this kind of situation in mind that abstract interpretation was introduced (COUSOT; COUSOT, 1977).

With abstract interpretation, 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. For example, a boolean value could yield `true`, `false` or `maybe` instead

```
1 local x = 0
2 while x > 0 do
3     x = x + 1
4 end
```

Code 23: Example of a situation in which a program can't be properly interpreted concretely

of its usual pair of possible values, and the abstract interpretation would yield `maybe` if the exact value couldn't be determined . Another example is reducing the possible range of values of an integer to the classes `positive`, `negative` and `zero`. While the first option creates an "escape route" with the `maybe` value, the second option allow convergence using the fixed-point technique.

In order to use the fixed-point technique in Code 23, we need to abstract the value of `x`. For instance, let's think in terms of signal (`positive`, `negative` and `zero`). Firstly, in line 1, `x` has the `zero` value. Then, at line 3, it becomes `positive` at the first iteration of the loop. We know that because any positive number added to zero returns a positive number and `1` is positive. In second iteration, we can be sure that `x` will remain positive, since two positives added together will always return another positive. Since we had two consecutive values of `x` that were equal, we have found a fixed point and we can stop iterating. The analysis would return that the final result of `x` would be `positive`, even if the final point of the program is never reached.

The main idea of abstract interpretation of programs is to interpret the program using abstract values. A concrete value is a value that a variable may have in a concrete semantics, such as integers, strings and arrays. An abstract value is a less precise representation of a concrete value. For example, for the concrete integer `5` we could associate an abstract value `positive` or `odd`. Each analysis defines an abstract domain to operate. These domains always have a similar form: Besides the set of possible abstract values, it has a bottom value, representing a value with no information, and a top value, representing a value that could be any of the abstract values. This form defines a lattice (GRäTZER, 1971) of the possible values that the variables could have.

Using the lattice model is clear why the fixed-point computation always end. Our lattices can have infinite elements, but must have a finite height. At the same time, as we execute the abstract interpretation, our variables assume values from the bottom to the top, but exclusively in this direction, since we can only make them less precise, as we gather new information. Since our lattice has finite height, we can be sure that at some point the program will terminate, even if assumes the top value for every variable.

The objective of abstract interpretation is to provide an abstract state for each program point in the analyzed program. Depending on the implementation, a program point can have different meanings. For instance, each program point could represent a statement or a basic block, in which statements are grouped together if there isn't any branching occurring between them. The abstract state is usually a mapping from every value in context to its abstract value.

We implemented the abstract interpretation with four functions. The main function is `AEval`, which abstractly interpret a program. This function is shared between all analyses and is independent of the abstract domain of each. It uses three auxiliary functions, `AExp`, `join` and `AVal`. These functions are redefined for each analysis.

The `AEVal` function is responsible for interpreting a program, returning an array of states, corresponding to each program point. The function takes 3 parameters: the program to be interpreted (represented in Pallene's internal representation), `AExp` and `join`. It returns an abstract state corresponding to the first argument (the program to be interpreted). Implementation wise, it looks a lot with a conventional interpreter. It handles each possible command in Pallene's internal representation, recursing in the command's children if necessary. In a `if` command, for example, it would recurse in the `condition`, `if` and `else`. Still in the `if` command example, `AEval` uses the `join` function after branching, to join the information gathered in the two possible paths. When it needs to compute the (abstract) result of an expression, it calls `AExp`.

We can see a fragment of a possible implementation of `AEval` in Code 24. In line 2, it creates the array of states, that will be returned in line 17. This array is updated in line 5 at each iteration of the algorithm, for the respective command evaluated. Before being returned, it is updated one last time with the final result of the evaluation, corresponding to the root command, that is usually a function command. In line 3, the initial state is created. It is passed as an argument to the interpretation in line 15. Also in line 15, we see the `go` function being called. This function is responsible doing the actual interpretation of each type of control flow nodes. We can see in lines 6 to 9 how it handles the `if` command. If the command to be interpreted is a terminal value (instead of being a control flow command, with children), it is treated in line 12, with the `AExp` expression, that will be described just after in this section.

Still in Code 24, note how the state is merged using the join function in lines 7, 8 and 9. In lines 7 and 8, we merge the resulting state of each branch with the current state of the interpretation. In line 9, we merge the two possible states into the current state. We will comment how the join function works later on in this section.

We have seen just now how `AEval` is implemented. Now see how to use it using our recurring example of abstracting integers in respect of their signal. We want to implement the three application functions, namely `AVal`, `AExp` and `join`.

`AVal` is usually the simplest of the three application functions to be implemented. It takes a concrete value and returns its abstracted counterpart. We can see a possible implementation for it in Code 25. It is quite simple, returning an abstract value for each possible signal that an integer can have. In this case, we didn't use the second argument, the state, but it can be useful in other situations.

`AExp` is a function that shows how to transform abstract values into other abstract values (for instance, what happens when we sum two positive(`+`) numbers). We can see a possible implementation in Code 26. It receives an expression command and the current state, and returns the equivalent state for the expression. We can comment on four of the many possible cases:

1. The first, in lines 2 to 4, takes a constant value and returns its abstract counterpart using the `AVal` function.

2. The second, in lines 5 to 7, returns the abstract value of a given variable in the current variable state. Note that the state only stores abstract values, so a call to `AVal` is unecessary.

3. The third, in lines 8 to 19, return the result of adding two abstract values in the integer signal domain. It matches each one of the possible cases that the values could have. For example, in line 10, if both values are undefined, then the result is also undefined. It is similar when both values can be any of the 3 possible classes, as in line 11, the result can be any integer. Then we can be more specific. In line 12 and 13, it handles the cases in which they are both zero or positive, as they will continue zero or positive, respectively. It continues for all relevant cases. If it doesn't match any case, for example when one value is positive and the other is negative, it returns the top value(`?`), meaning that the result can be any possible integer.

4. Lastly, in line 23, if the requested expression isn't any of the treated cases, it simply return the current state without any modifications.

The `join` function receives two abstract program states and returns a new one, considering the information contained in both states. In Code 27 we

```lua
function AEval(root_cmd,AExp,join)
  local states = {}
  local init = {}
  local function go(cmd,state)
    states[#states+1] = state
    if cmd.tag == 'if' then
        local state_then = join(AEval(cmd.then_,state),state)
        local state_else = join(AEval(cmd.else_,state),state)
        state = join(state_then,state_else)
    elseif cmd.tag ...
    ...
    return AExp(cmd,state)
  end
  -- initial state is empty
  local root = go(root_cmd, init)
  states[#states+1] = root
  return states
end
```

Code 24: Fragment of the `AEval` implementation in Lua

can see an example of this function implemented. The Code has two function definitions, `join_val`, between lines 1 to 12 and `join`, between lines 14 to 24.

Let's start by looking at the `join` function. It is fairly simple, it iterates between each state and finds a new value for each of the variables, by merging their values of each state with the `join_val` function. In this particular case, the order of the iterations is irrelevant, mostly because `join_val` should be commutative for this domain. Also, note that `join_val` should be indepotent for this domain. However, there are domains in which these properties are not expected. There are cases in which the `join` function has semantics similar to assignment, with the second parameter overwriting the information from the first.

The `join_val` receives two abstract values and return a new one, with their information merged. In this case, it is straightforward: When the two states agree on a value, as in line 6, we use this agreed value. In lines 8 and 9, we can see the cases that in one path one of them was undefined, but it was defined in the other path. In these cases, we can return that the result may have the defined value (depending on the analysis, it can be useful to return the set containing both undefined and the defined value). But if one path says that a certain value is for sure negative, while another says that it is for sure positive, we can only return that it can be both (represented with the top value (?)).

A final consideration is in the order of the program traversal performed by `AEval`. It always follows the same pattern, for example, in a `if` statement, it always checks the `condition`, then the `if` branch and lastly the `else` branch. This

```
function AVal(value, state)
    if     exp.value == 0 then return "0"
    elseif exp.value >  0 then return "+"
    elseif exp.value <  0 then return "-"
    end
end
```

Code 25: Example of implementation of the `Aval` function for integers over the abstract domain of signals

```
function AExp (exp,state)
    if exp.tag == 'const'   then
        -- from concrete to abstract
        return AVal(exp.src1,state)
    elseif exp.tag == 'localvar' then
        -- states already are abstract
        return state[exp.src1.id]
    elseif exp.tag == '+'   then
        local lhs,rhs = AExp(exp.src1, state), AExp(exp.src2,
            state)
        if lhs == '_' or rhs == '_' then return '_' end
        if lhs == '?' or rhs == '?' then return '?' end
        if lhs == '0' and rhs == '0' then return '0' end
        if lhs == '+' and rhs == '+' then return '+' end
        if lhs == '+' and rhs == '0' then return '+' end
        if lhs == '0' and rhs == '+' then return '+' end
        if lhs == '-' and rhs == '-' then return '-' end
        if lhs == '-' and rhs == '0' then return '-' end
        if lhs == '0' and rhs == '-' then return '-' end
        return '?'
    elseif cmd == '-'   then
        ...
    else
        return state
    end
end
```

Code 26: Example of the implementation of the `AExp` function for integers over the abstract domain of signals

```lua
local function join_val(a,b)
    -- nil means '_' (undefined)
    a = a == nil and '_' or a
    b = b == nil and '_' or b
    -- when both are _, ? or same value
    if a == b                then return a end -- or b
    -- when one is undefined, define it
    if a == '_' and b ~= '_' then return b end
    if b == '_' and a ~= '_' then return a end
    -- if the values differ, nothing can be said
    return '?'
end

local function join(state1,state2)
    local ret = {}
    for k,v in pairs(state1) do
        ret[k] = join_val(state1[k],state2[k])
    end
    -- even if a given key can be recalculated,
    -- its value shouldn't change.
    for k,v in pairs(state2) do
        ret[k] = join_val(state1[k],state2[k])
    end
    return ret
end
```

Code 27: Example of the implementation of the `join` function (and its auxiliary `join_val` function) for integers over the abstract domain of signals

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 in this chapter, we will be using this order to describe some of our procedures. We can note this ordering being used to build the array in Code 24, on lines 5 and 16.

4.2 Evaluating our results

We used the same set as described in Section 2.3.1 plus the four new benchmarks introduced to evaluate records, as described in Section 3.5. In these sections we have already described the objectives of each program. In this section, we will give more details on their implementation. These details can be useful when we discuss the results of the implemented optimizations.

We adjusted the parameters in order to reduce the noise of the final result, similarly to the other moments in this work that we evaluated run time. Likewise, we used one second of run time for each of the 10 executions as a minimum, but some benchmarks would take several seconds to run even with the smallest inputs. We measured the run time using the "perf" utility and used the same system configuration as before.

1. **binarytrees:** it receives a positive integer N and builds a binary tree of height N, forming 2^{N+1-1} array nodes. Each node has only two elements, its children, and no extra values, and leaves are an array of two `falses`. After mounting the tree, it's consistency is checked by a breadth first search. Then, a second loop recreates and rechecks recursively the tree, for sequential heights, up to N. This benchmark is focused on memory allocation and garbage collection.

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. This benchmark calls Pallene from Lua just once, but the called function, in Pallene, calls the other internal Pallene function several times, in a tight loop.

4. **centroid:** it creates N points and finds the centroid between them. While the creation of the points is a Pallene function called repeatedly from Lua, the function to calculate the centroid is called just once from Lua. This function that calculates the centroid does a quadratic number of operations. In this benchmark, data is bridged frequently between the languages through the shared runtime. Memory allocation, array accesses, assignments and arithmetic expressions are frequent.

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. This benchmark uses heavily array of arrays of strings, allocating, modifying and accessing them. The strings could be simply enumerations since they only denote the state of the cells. However, the cells containing strings is useful to us, because we can then investigate their operations. The code frequently calls Pallene from Lua and uses the returned values in Lua.

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 called "pancake(fannkuch) flipping", in which the elements of the vector are swapped in position until the first position contains the number 1.

By the problem definition, this procedure is executed at most $N * log(N)$ times. It also keep track of a checksum for each permutation in order to verify the correctness of the algorithm. With an original array and a working copy, this benchmarks does few allocations but a lot of memory manipulation. It is executed almost exclusively by Pallene.

8. **fasta:** it calculates the similarity between strings representing DNA sequences. It uses a lot of string substitutions and linear searches on arrays. Note that string substitution is a built-in function in Pallene that calls the corresponding function in the Lua context, it is not a function in C compiled alongside the emitted C code.

9. **mandelbrot:** it generates the mandelbrot set of a given integer, N. It outputs an image in the netpbm format. It does a handful of arithmetic operations for each of the quadratic iterations of the algorithm. It prints each point in order, after calculating it, so it doesn't allocate heap memory, using only integer values.

10. **objmandelbrot:** it is similar to the mandelbrot benchmark but uses arrays instead of pairs of integers. It stresses the communications between Pallene and Lua, because it calls Pallene functions very frequently from Lua, receiving and handling back the data. It calls Pallene functions a quadratic number of times. Each Pallene function is a simple arithmetic operation, but they always return the value as a newly allocated array.

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. In this benchamark, all Pallene functions returns the defined record and all of them receive one or two records, with the exception of the `new` function, that takes two `float` s and return a record with both of them. All the functions return a newly created record.

12. **matmul:** it multiplies a $N \times N$ matrix with itself a number of times. It accesses and multiplies the values in the array of array of floats a quadratic times in Pallene. The data is created in Lua and then the data is passed as an argument to the Pallene function, that is called repeatedly from Lua.

13. **nbody:** it computes the solution to the n-body problem. The problem consists on 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. It heavily uses arithmetic operations on the body's

fields. For each step, the number of operations performed is quadratic in respect to the number of bodies, but the number of bodies is fixed, so the algorithm is linear.

14. **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. It is a quadratic algorithm executed almost exclusively in the Pallene side.

15. **sieve:** it computes the sieve of Eratosthenes for a given positive integer N, calculating all the prime numbers up to N. It is a simple quadratic algorithm that uses a single N-sized array in-place to perform its operations.

16. **spectralnorm:** it calculates the spectral norm of a $N \times N$ square matrix for a given N. It allocates up to N N-sized arrays. Lua calls the Pallene function only once and the Pallene function is linear.

4.3 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 caller, many optimization 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 duplicate 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.

4.3.1 Implementation

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, as described in the Section Abstract Interpretation. 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 subsequently 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. In Pallene's internal representation, local variables are identified by integers and their information (such as name or type) is stored alongside with the function, in an array indexed by the variable identifier. However, the compiler doesn't provide any guarantees about these identifiers: they could start at any number, can have holes in the sequence and there could be any order on them. Without these guarantees, it was quite hard to predict when there would be an identifier conflict.

In order to facilitate the implementation, we introduced the idea of the normal form for the internal representation. This normal form guarantees that local variables indexes starts at one, that the indexes are sequential (without holes) and that their numbering order has to be in order of appearance (following the convention of our abstract interpretation functions). We implemented a function that verifies whether a code were in normal form and another function that would transform it to the new form.

To avoid conflict between the variables of the function to be inlined and the called, we adopted a simple strategy: With the guarantees provided by the normal form (both functions would be already normalized), we simply calculate the largest identifier in the outer function and add the internal indexes to this value. At the end of inlining this particular call, we would put the caller function in normal form again, and then proceed to the next call.

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. 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 this 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.

Summing it up, the inlining process of a call to `B` into function body `A` has the following steps:

1. Find the function's highest variable index (`top`) of `A`. it is important to recalculate it at each new inline, because it can change.

2. Transform each argument passed from `A` into an assignment to a new variable. This new variable will have its index equal to its original identifier plus the value of `top`.

3. Do a first depth-first pass on `B`, adding `top` to each variable index. The changes occur in-place.

4. Do a second depth-first pass on `B`, transforming each `return` into assignments, creating local variables when appropriate. During this pass, the loop commands (`while`, `for`) are labeled using a simple sequence of integers convention.

5. Clean the result of the last step, removing label annotations.

6. Create a `sequence` command node with the results of the two passes and substitute the call for `B` in `A` for the new command.

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.

4.3.2 Limitations

Inlining is performed whenever it's possible. We know that it can introduce overhead, because it essentially increases code size. We could implement heuristics to mitigate this possible overhead, possibly based on the cumulative number of instructions of the called function.

4.3.3 Results

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 expressive results, between 12% and 18%. Fasta, queen and spectralnorm benchmarks also yielded significant results, close to 10%. We can see all results on Table 4.1 and can see them summarized in Figure 4.1.

4.4 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. This could enable some aggressive C-level optimizations. It

Benchmark	Control run time avg. (sec)	Number of calls inlined	Run time avg. after function inline (sec)	Difference in run time (%)
binarytrees	3.63	6	2.98	18
recordbinarytrees	4.86	6	4.28	12
binsearch	4.82	1	4.72	2
centroid	1.51	0	1.51	0
recordcentroid	2.20	0	2.20	0
conway	4.80	6	4.22	12
fannkuchredux	4.82	0	4.82	0
fasta	2.87	3	2.61	9
mandelbrot	7.77	0	7.77	0
matmul	9.67	0	9.67	0
nbody	3.35	1	3.22	4
recordnbody	8.30	1	7.97	4
objmandelbrot	9.83	6	9.24	6
recordobjmandelbrot	14.09	6	13.24	6
queen	2.30	3	2.09	9
sieve	28.94	0	28.94	0
spectralnorm	27.19	4	25.01	8

Table 4.1: Performance gains of performing the function-inlining optimization

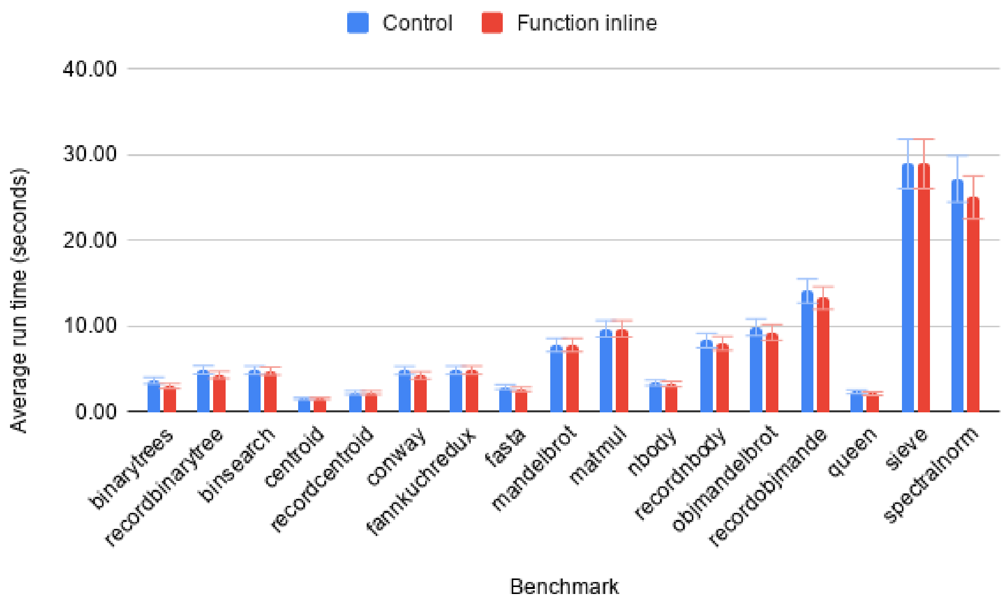

Figure 4.1: Average run time after 10 executions when performing the function-inlining optimization

is recommended that this optimization is done after function inlining, as described beforehand on Chapter 3.

4.4.1 Implementation

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.

4.4.2 Limitations

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 handle 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).

4.4.3 Results

We applied the optimization on programs with function inline already performed. The results discussed in this section follows 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 behaviour that the compiler would execute.

This optimization has shown expressive results, usually in the range of 13% to 55%, as we can see in Table 4.2. In particular, we can see that in the three benchmarks in which LuaJIT performed better (centroid, matmul and objmandelbrot), we achieved good speed-ups. We can see this summarized in figure 4.3.

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 6 "possible" cases. LuaJIT can optimize these cases since it does its analysis in runtime, calculating the traces alongside the interpretation.

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 a 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. We can see the results summarized in figure 4.2.

Benchmark	Control run time (with function inline) avg. (sec)	Number of scalar accesses opportunities	Number of scalar accesses replaced	Run time avg. after scalar replacement (sec)	Difference in run time (%)
binarytrees	2.98	3	3	2.81	5.34
recordbinarytrees	4.28	0	0	4.28	0
binsearch	4.72	0	0	4.72	0
centroid	1.51	3	3	1.21	19.64
recordcentroid	2.20	0	0	2.20	0
conway	4.22	3	3	3.65	13.46
fannkuchredux	4.82	1	0	3.33	0
fasta	2.61	2	0	2.61	0
mandelbrot	7.77	0	0	7.77	0
matmul	9.67	6	1	8.21	15.10
nbody	3.22	44	21	2.81	12.46
recordnbody	7.97	0	0	7.97	0
objmandelbrot	9.24	0	5	4.12	55.41
recordobjmandelbrot	13.24	0	0	13.24	0
queen	2.09	6	0	2.09	0
sieve	28.94	1	0	25.03	13.51
spectralnorm	25.01	9	2	15.90	36.43

Table 4.2: Performance gains of performing the scalar replacement of aggregates optimization

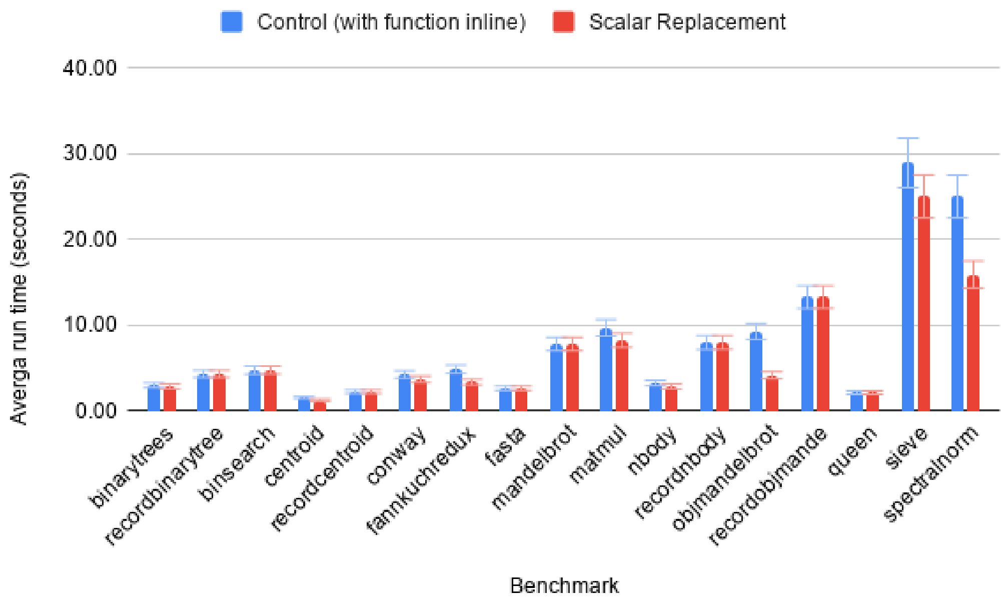

Figure 4.2: Average run time after 10 executions when performing the scalar replacement of aggregates optimization

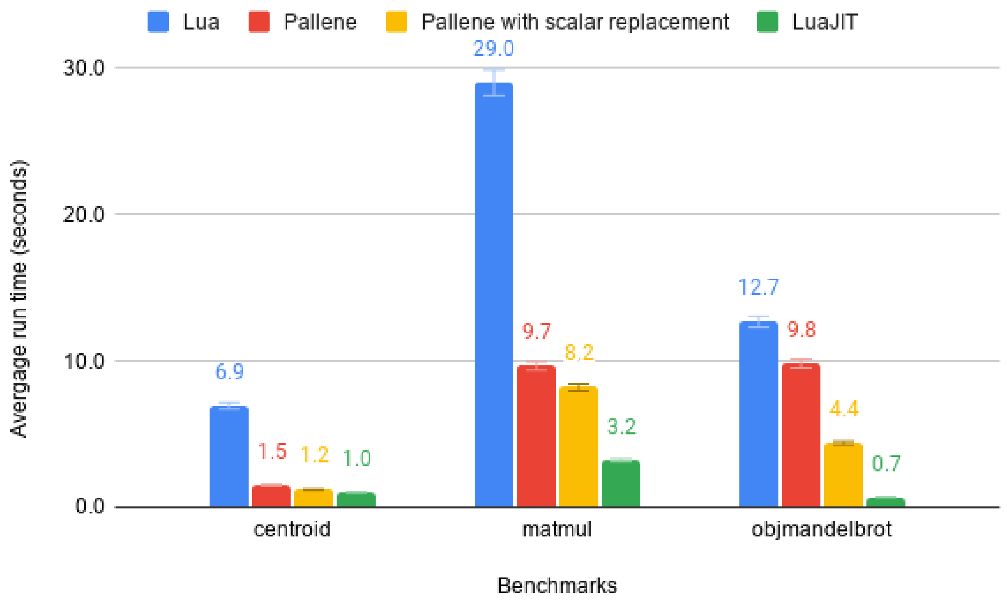

Figure 4.3: Comparison of the gains in the benchmarks discussed in Chapter 2

4.5 Renormalize

When accessing an array, 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 growing 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 a 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.

4.5.1 Changes in the Pallene compiler's internal representation

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.

4.5.2 Implementation

Our algorithm is straightforward: we collect the information for every `NormalizeArr` in the internal representation and apply the transformations if it matches one of the three cases explained in Chapter 3.

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 have 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 `for`s.

4.5.3 Limitations

The presence of function calls in the function body effectively kills the optimization. This is mitigated if there were a previous function inline pass.

4.5.4 Results

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

We can see in Table 4.3 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. The results are also summarized in Figure 4.4.

4.6 Initialization

After an array or a record initialization, there are some checks that could be removed with certain conditions met, as we discussed in Chapter 3. These includes a renormalization call and a check for garbage collection. Since renormalization calls were already discussed, we will be talking only about the `condGC` removals.

Benchmark	Control run time (with function inline) avg. (sec)	Number of renormalized calls unified	Number of renormalized calls hoisted	Run time avg. after the renormalize removals (sec)	Difference in run time (%)
binarytrees	2.98	5	0	2.82	5.23
recordbinarytrees	4.28	1	0	4.24	0.91
binsearch	4.72	0	1	4.49	4.88
centroid	1.51	3	1	1.38	8.87
recordcentroid	2.20	0	0	2.20	0
conway	4.22	16	22	3.24	23.34
fannkuchredux	4.82	10	6	4.09	15.18
fasta	2.61	5	3	2.16	17.27
mandelbrot	7.77	0	0	7.77	0
matmul	9.67	3	6	8.88	8.22
nbody	3.22	60	6	2.57	20.19
recordnbody	7.97	0	3	7.42	6.94
objmandelbrot	9.24	10	0	8.56	7.4
recordobjmandelbrot	13.24	0	0	13.24	0
queen	2.09	0	3	1.95	6.9
sieve	28.94	0	5	26.34	8.98
spectralnorm	25.01	1	6	23.06	7.8

Table 4.3: Performance gains of performing the removal of renormalizes optimization

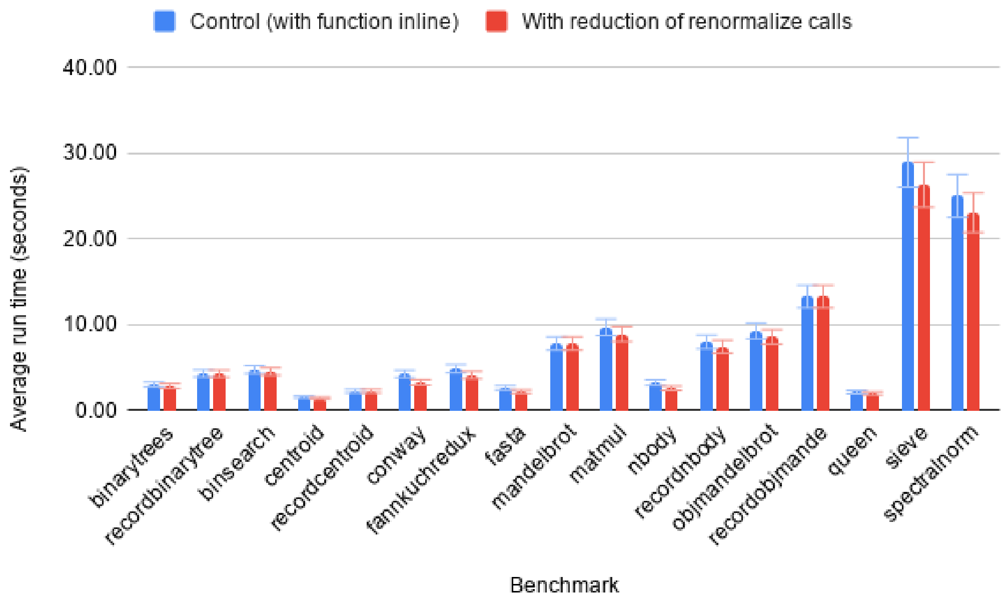

Figure 4.4: Average run time after 10 executions when performing the removal of renormalizes optimization

Benchmark	Control run time (with function inline) avg. (sec)	Number of condGC commands removed	Run time avg. after the removals (sec)	Difference in run time (%)
binarytrees	2.98	3	2.96	0.69
recordbinarytrees	4.28	3	4.24	0.78
binsearch	4.72	0	4.69	0.67
centroid	1.51	2	1.50	0.87
recordcentroid	2.20	2	2.18	0.95
conway	4.22	2	4.18	0.93
fannkuchredux	4.82	4	4.79	0.67
fasta	2.61	4	2.59	0.88
mandelbrot	7.77	2	7.74	0.40
matmul	9.67	2	9.61	0.66
nbody	3.22	1	3.18	1.00
recordnbody	7.97	1	7.91	0.79
objmandelbrot	9.24	1	9.19	0.50
recordobjmandelbrot	13.24	1	13.16	0.67
queen	2.09	1	2.08	0.86
sieve	28.94	2	28.66	0.98
spectralnorm	25.01	3	24.77	0.99

Table 4.4: Performance gains of performing the removal of condGC optimization

4.6.1 Implementation

We implemented the heuristic described in Chapter 3: For each basic block, it checks for the presence of `condGC` calls, and if at least one is present, we apply the transformation on the block. The transformation consists of removing all of `condGC` calls and adding one new at the end of the block. We implemented this with a simple depth-first traversal.

4.6.2 Results

While we could find significant results (around 10%) in an example with a tight loop with just an array creation, we found it to be too artificial. In all of the benchmarks, the results couldn't even be measured properly, being always less than the standard deviation that perf calculates. Nevertheless, we present the results in table 4.4. Note that the values of the last column are percentages below 1%.

4.7 Function Headers

In the C code emitted by Pallene, every Pallene function has a wrapper that is exposed to Lua (the respective entry point function). In this wrapper, a variable is calculated and passed to the wrapped function, independently if the

Benchmark	Control run time (with function inline) avg. (sec)	Number of G calculations removed	Run time avg. after the removals (sec)	Difference in run time (%)
binarytrees	2.98	4	2.84	4.48
recordbinarytrees	4.28	4	4.08	4.64
binsearch	4.72	3	4.57	3.24
centroid	1.51	3	1.46	3.39
recordcentroid	2.20	3	2.12	3.48
conway	4.22	6	3.93	7.02
fannkuchredux	4.82	2	4.71	2.18
fasta	2.61	3	2.52	3.48
mandelbrot	7.77	2	7.61	2.08
matmul	9.67	2	9.46	2.20
nbody	3.22	6	3.00	6.72
recordnbody	7.97	6	7.41	6.96
objmandelbrot	9.24	7	8.54	7.56
recordobjmandelbrot	13.24	7	12.25	7.49
queen	2.09	2	2.05	2.16
sieve	28.94	2	28.36	2.00
spectralnorm	25.01	2	24.23	3.14

Table 4.5: Performance gains of performing the removal of condGC optimization

wrapped function will actually use it. We have discussed this in details in the last chapter. In a tight loop in Lua that calls an identity function in Pallene, removing these calculations showed a reduction of 5% in the overall time.

4.7.1 Implementation

When emitting the C code, we make a simple introspection, using a depth-first search, for the commands that use userdata: dealing with upvalues, strings, records or functions-as-values. If neither of these commands is present, we can safely remove the calculations (passing a `NULL` pointer to the function instead of calculating the value).

4.7.2 Results

As with the Initialization optimizations, the cases in which this optimization is relevant are quite artificial. In our benchmarks, it only had reductions in the cases where a benchmarking function would call a Pallene function repeatedly, for example in objmandelbrot, even in those cases, the gains were quite small. We can see the results in Table 4.5.

4.8 Array vs Records

We have seen in the last chapter that using records instead of arrays can lead to significant overhead. This is specially true when creating arrays at the Pallene side. We implemented two optimizations to deal with this issue.

4.8.1 Modifying the operation new_key

When Pallene creates records, the function `new\_key` is called. This function creates a new field for a given Lua table. For Lua, the key can have any type, so it performs a large amount of verifications. All of them are unnecessary knowing the type of the key beforehand (specially knowing that it will be a string) and knowing that the record had just been created. By replacing this function call for a call to a more specific version of it, we saw expressive improvements in run time. In special, in the binary trees example, in which records are created exponentially, we saw it becoming faster than the array implementation. This makes some sense, since we guarantee that records will always be the same size and with the same fields, in opposition to arrays that could potentially have any size. It is important to note that was necessary to add some macros and functions that were in the Lua implementation (but invisible to Pallene) into the emission of the C code.

The results for this optimization were good, ranging from 15% to 30%. It is interesting to note that this was the range of overhead for records observed in the last Chapter 3. We can see the results in Table , we present the results for all benchmarks, even with this optimization only being appliable to the benchmarks that use records. We also present a comparison of the use of this optimization with the array counterpart of each record benchmark, in Figure 4.5, showing that for the binarytrees and centroid cases we effectively removed the discussed overhead and that for the objmandelbrot and nbody, we reduced the gap between the use of the two kinds of key.

4.8.2 String hashing

Another path for optimizing the use of records was to optimize the obtainment of memory slots for records. To get the memory slot associated with a given (string) key, it uses the `pallene\_getstr` function, that searches in a linked list of strings for the appropriate slot. This was implemented in this way because of Lua's random seed for string hashing. We experimented fixing the result of the seed function to `0` and modifying the code accordingly.

Benchmark	Control run time (with function inline) avg. (sec)	Number of new_key calls re-implemented	Run time avg. after the re-implementations (sec)	Difference in run time (%)
binarytrees	2.98	0	2.98	0
recordbinarytrees	4.28	4	2.96	30.9
binsearch	4.72	0	4.72	0
centroid	1.51	0	1.51	0
recordcentroid	2.20	4	1.66	24.57
conway	4.22	0	4.22	0
fannkuchredux	4.82	0	4.82	0
fasta	2.61	0	2.61	0
mandelbrot	7.77	0	7.77	0
matmul	9.67	0	9.67	0
nbody	3.22	0	3.22	0
recordnbody	7.97	19	5.48	31.25
objmandelbrot	9.24	0	9.24	0
recordobjmandelbrot	13.24	2	11.22	15.26
queen	2.09	0	2.09	0
sieve	28.94	0	28.94	0
spectralnorm	25.01	0	25.01	0

Table 4.6: Performance gains of performing the new_key implementation change optimization

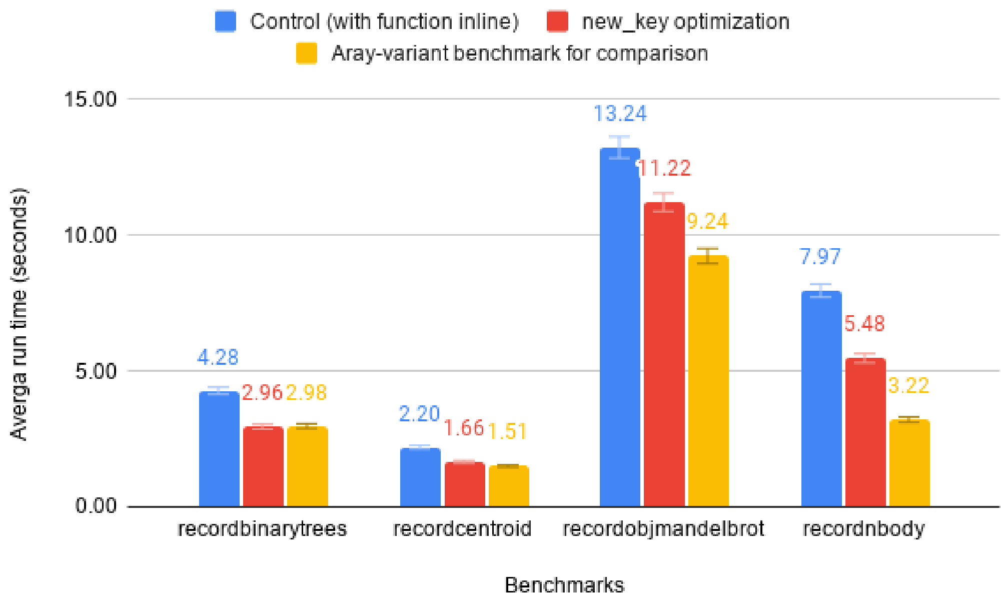

Figure 4.5: Comparison of run time of the benchmarks that use records with and without our optimization and their array counterpart.

Benchmark	Control run time (with function inline) avg. (sec)	Number of new_key calls re-implemented	Run time avg. after the re-implementations (sec)	Difference in run time (%)
binarytrees	2.98	0	2.98	0
recordbinarytrees	4.28	4	2.71	8.21
binsearch	4.72	0	4.72	0
centroid	1.51	0	1.51	0
recordcentroid	2.20	4	1.50	9.47
conway	4.22	0	4.22	0
fannkuchredux	4.82	0	4.82	0
fasta	2.61	0	2.61	0
mandelbrot	7.77	0	7.77	0
matmul	9.67	0	9.67	0
nbody	3.22	0	3.22	0
recordnbody	7.97	19	4.62	15.61
objmandelbrot	9.24	0	9.24	0
recordobjmandelbrot	13.24	2	9.87	12.03
queen	2.09	0	2.09	0
sieve	28.94	0	28.94	0
spectralnorm	25.01	0	25.01	0

Table 4.7: Performance gains of performing the string hashing optimization

With the seed being fixed, we can know the position of the memory slot in compile time. We can then replace the `pallene\_getstr` calls for the value directly.

This optimization further decreased the execution time of the tested benchmarks, with reductions of 8% to 15%. However, this optimization requires a modification on the Lua interpreter that Pallene uses. Since one of the objectives of Pallene is to use an unmodified Lua interpreter, this would be another layer of friction. That being said, further investigations of this optimization could lead to proposal of changes to PUC-Lua itself.

4.9 Conclusion

We have seen that our optimizations were effective. We can now present a new consolidated overview of our results, in both table form (in Table 4.8) and figure form (Figure 4.6). In these results we show the composed results of all the optimizations that we found interesting to be implemented: function inlining, reduction of renormalize calls, scalar replacement of aggregates, the new implementation of new_key and the changes in the function header of the entry-point functions.

We achieved decreases in total run time from 8% to 62%, with an average of 34%. In particular, we have made progress in reducing the overhead of using records and we have shortened the gap between Pallene and its alternatives, specially LuaJIT.

Benchmark	Control (sec)	Theoretical avg run-time (sec)	Theoretical gains (%)	Empirical avg run-time (sec)	Empirical gains (%)	Difference in gains (overhead)
binarytrees	3.63	3.18	12.31	3.49	3.90	8.41
recordbinarytrees	4.86	3.08	36.58	3.11	36.01	0.57
binsearch	4.82	4.22	12.44	4.63	3.86	8.58
centroid	1.51	1.10	27.31	1.15	23.53	3.78
recordcentroid	2.20	1.51	31.41	1.55	29.76	1.65
conway	4.80	2.30	52.18	2.43	49.41	2.77
fannkuchredux	4.82	2.60	46.14	2.72	43.55	2.59
fasta	2.87	2.26	21.11	2.43	15.19	5.92
mandelbrot	7.77	7.01	9.80	7.55	2.85	6.95
matmul	9.67	3.55	63.33	3.76	61.16	2.16
nbody	3.35	1.18	64.78	1.28	61.85	2.92
recordnbody	8.30	4.52	45.57	4.77	42.58	2.99
objmandelbrot	9.83	3.38	65.64	3.67	62.62	3.02
recordobjmandelbrot	14.09	10.14	28.07	10.25	27.28	0.79
queen	2.30	2.05	10.67	2.11	8.17	2.50
sieve	28.94	12.12	58.13	13.38	53.78	4.35
spectralnorm	27.19	12.02	55.81	12.14	55.37	0.44

Table 4.8: Performance gains of performing all the optimizations

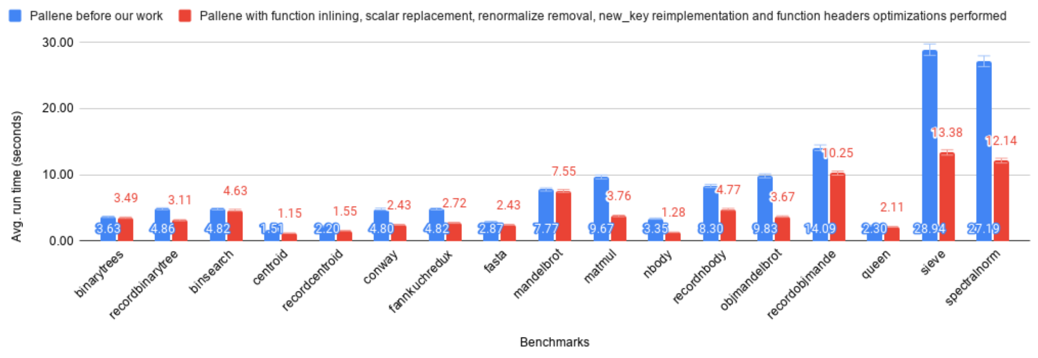

Figure 4.6: Average run time after 10 executions when performing all optimizations

We can also evaluate the overhead of compounding the optimizations. We calculated the theoretical gain by multiplying the gains of each optimization into the control, finding the second and third columns of Table 4.8. Then, we actually applied each optimization implementation in order to the benchmarks, finding results slightly different from the theoretical ones. We can see that the overhead is significant in the binarytrees and the binarysearch benchmarks, but negligent overall, with an average of 3.55%.

5 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 great 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 behaviours present in Lua, such as meta programming features. By guaranteeing the absence of these behaviours, 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 behaviour 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 in Chapter 2, 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 aren'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 have also seen that there are cases in which Pallene is slower than one of its alternatives (LuaJIT), indicating that there are some optimizations that weren't performed in Pallene.

We end up with four opportunities of optimization:

- Replacing arrays for its scalar components, which can't be properly delegated to the C compiler.
- Removing unnecessary calls that exist only in C but are invisible to the Pallene user, such as array's renormalizations.
- Removing unnecessary calculations at the entry point function that bridges Lua and Pallene code.
- Optimizing table access with string typed keys.

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, specially 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 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 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. Most of the benchmarks benefited from this optimization, with run time reductions varying between 16% to 56%.

The next optimization implemented was the reductions in renormalize calls. A certain property of the renormalize function allow 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%.

Our next work was on initialization of data structures. After creating a data structure, Pallene always verify its size and the status of the garbage collector. These checks could be removed if enough information of the context is known. While removing these checks inside a tight loop would lead to reductions of 10%, these optimizations didn't yield significant results in the benchmarks.

Lastly, we investigated why records are slower than arrays in Pallene. We implemented two optimizations: one in the key creation, exploiting the information of the type system, leading to reductions of 15% to 30% in the four relevant cases, and another on the string value acquisition, by fixing the hash function to get the string more predictably, which showed reductions from 8% to 16%. However, this last optimization can't be performed without modifications on the Lua interpreter. These modifications compensated for the overhead introduced by the records in half of the benchmarks and mitigated it greatly in the other half.

By applying all optimizations as the compiler would do, we have seen reductions of 10% to 66% in total run time.

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.

6
Bibliography

COUSOT, P.; COUSOT, R. 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**. Los Angeles, California: ACM Press, New York, NY, 1977. p. 238–252.

GCC. **Declaring Attributes of Functions**. 2021. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html>.

GCC. **Options That Control Optimization**. 2021. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

GCC. **Other Built-in Functions Provided by GCC**. 2021. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>.

GRäTZER, G. **Lattice Theory. First Concepts and Distributive Lattices**. [S.l.: s.n.], 1971.

GUALANDI, H. M. **The Pallene Programming Language**. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2020.

GUALANDI, H. M.; IERUSALIMSCHY, R. Pallene: a statically typed companion language for lua. In: CAMARÃO, C.; SULZMANN, M. (Ed.). **Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP 2018, Sao Carlos, Brazil, September 20-21, 2018**. ACM, 2018. p. 19–26. Disponível em: <https://doi.org/10.1145/3264637.3264640>.

GUALANDI, H. M.; IERUSALIMSCHY, R. Pallene: A companion language for lua. **Science of Computer Programming**, Elsevier, p. 102393, 2020.

IERUSALIMSCHY, R. Lua performance tips. **Last update: Wed Apr 27 09: 04: 45 BST 2016 (build 83)**, p. 15, 2008.

JAMBOR, M. The new intraprocedural scalar replacement of aggregates. **GCC Summit**, 2010.

MUCHNICK, S. S. **Advanced Compiler Design and Implementation**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN 1558603204.

PALL, M. **Allocation Sinking Optimization**. 2012. Disponível em: <http://wiki.luajit.org/Allocation-Sinking-Optimization>.

RICE, H. G. Classes of recursively enumerable sets and their decision problems. **Transactions of the American Mathematical Society**, American Mathematical Society, v. 74, n. 2, p. 358–366, 1953. ISSN 00029947. Disponível em: <http://www.jstor.org/stable/1990888>.

WEGMAN, M.; ZADECK, K. Constant propagation with conditional branches. In: . [S.l.: s.n.], 1985. p. 291–299.