Targeted Lua Optimization Using Terra

TOFE ALIMI, Stanford University, USA

This project focused on designing a higher level interface by which we can target specific paradigms that appear in Lua code and translate them into optimized Terra, a low-level systems language metaprogrammed in Lua, in order to improve runtime performance. Specifically, since Terra sits in a separate, JIT-compiled environment from the surrounding Lua [2], we are able to localize more CPU-intensive code to generated Terra functions and are still able to maintain the higher level interface and abstractions provided by Lua. While our approach and interface could be improved in future iterations, we focused specifically on functions composed of either loops or various recursive calls that handle primitive types, such as boolean and number (integer) types. Upon transposing these high-target Lua functions into Terra counterparts we saw significant runtime improvements on the order of 10-15X. Ideally, we would be able to translate all Lua into Terra to achieve such performance increases across all Lua programs, but this is simply unfeasible given the higher level abstractions Terra simply cannot offer, or would be impractical to implement. Nevertheless, perhaps in the future, our module will be able generate optimized Terra code for functions handling higher level types.

Additional Key Words and Phrases: Lua, Terra, JIT-compilation, interpreted language, optimization

ACM Reference Format:
Tofe Alimi. 2010. Targeted Lua Optimization Using Terra. *Proc. ACM Hum.-Comput. Interact.* 9, 4, Article 39 (March 2010), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 BACKGROUND

Lua is already one of the fastest and most lightweight scripting languages on the market today. Its clean interface with low-level abstractions, particularly through the C API, have made Lua even more desirable. Where Terra comes into play is that it provides an entirely separate environment metaprogrammed in Lua for generating low-level code which is often times either more versatile or easy to use than interfacing directly with the C API [1]. What this means is that we can extract CPU-intensive operations, isolate them within Terra function which are JIT-compiled at runtime offering to presumably increase performance due to both LLVM optimizations [3] and the speed of compiled code itself; as long as we are able to generate a valid representation in Terra, given its lower-level abstractions, we should be able to see performance increases. Hence, this project focuses on automating this process of generating optimized Terra code; alternatively, we could have generated optimized C, leveraging Lua's C API [1], though the various stack calls and necessary dependencies and interaction between Lua and C may have yielded undesirable performance. In the end, Terra's nearly independent runtime seemed optimal for this task [2]. One of the main themes we discussed in class was the trade-offs between the three different programming languages: scripting, functional, and systems. Essentially, this project delves into this theme by exploring how we can leverage Lua and Terra, as well as the underlying frameworks enabling this interoperability, namely LuaJIT and LLVM [2], to find a balance between ease-of-use, through higher level abstractions, semantics, etc., and more controlled memory-usage and performance.

Author's address: Tofe Alimi, Stanford University, 450 Serra Mall, Stanford, CA, 94305, USA, oalimi@cs.stanford.edu.

2573-0142/2010/3-ART39 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 APPROACH

There were two main steps necessary to modularizing code optimization: the first was identifying target functions, and the second was translating Lua syntax to Terra. After doing so, we prepared a suite of cases against which we could compare performance of original Lua and optimized Terra source code.

2.1 Target Function Identification

High-target functions were identified primarily on those containing inner loops or recursive calls - as defined per the scope of our problem. To do so, we first parse Lua source code into an AST utilizing *lua-parser* [6]. For example, take the following code snippet:

```
function simple_add(x)
    for i=1,10 do
        x = x + i
    end
    return x
end
print(simple_add(10))
```

which, when run through the parser, yields the the following AST:

```
{ `Set{ { `Id "simple_add" }, { `Function{ { `Id "x" }, { `Fornum{ `Id "i",
`Number "1", `Number "10", { `Set{ { `Id "x" }, { `Op{ "add", `Id "x", `Id "i" }
} } } }, `Return{ `Id "x" } } } }, `Call{ `Id "print", `Call{ `Id "simple_add",
`Number "10" } } }
```

Upon parsing our source code, we first identify the `simple_add` function which has a for-loop within making it a desirable target function for optimization. Since Terra does not provide higher-level abstractions like Lua tables, we must then ensure that the calls made within the function are available to the Terra interface alone. Although Lua functions, as well as external variables, can be incorporated into Terra functions by means of casting and escaping, respectively, for the purposes of this project, we only generate Terra code for more "primitive" Lua functions. Besides standard math library functions which can easily be translated to their C counterparts in Terra, if external functions, other than the scoping function are called, or table operations are performed, the function will not be selected for optimization.

The reason for this is that we quickly found that interfacing higher level Lua structures in Terra is a very tedious process. While we could translate some Lua tables, that effectively serve as lists, to arrays in Terra, our first attempts to implement Lua tables, the language's defining structure, as dictionaries/maps in Terra was a far more extensive process than we would have hoped. Furthermore, type-checking/inference of these structures added an additional level of difficulty (as will be discussed in the following section), so we narrowed the scope of the accepted functions to those with only primitive usage.

2.2 Terra Translation

One of the major differences between Lua and Terra syntactically is that Terra implements static type checking as opposed to Lua's dynamic type checking [8] which means that type annotations are required. One of the reasons for a more "primitive" function identifier is that we must implement a basic type-inference module. To do so, we first identify all variables within the scope of the function, including arguments. We then traverse the abstract syntax tree to search for operations (tagged as `Op`) performed on the noted variables to help identify the variable type. A similar identification scheme is used across the different types. In the case above, an `add` operation is

performed on x by a defined int type, i, so we know x must be an integer. We are thus able to generate a new Terra function:

```
terra simple_add(x : int)
    for i=1,10 do
        x = x + i
    end
    return x
end
print(simple_add(10))
```

Unfortunately, if any variables scoped within the function do not have tagged types by the end of the traversal, the function is not optimized - safe, non-optimized code is better than botched "optimized" code due to incorrect type inference!

In the case of Lua tables, there is no consistent typing across keys and values. A single table can have keys that are boolean, string, and integer types, and even have functions as values. This makes it far harder to identify the *true* type of a table. We initially tried to incorporate Lua tables as arrays of enums of booleans and integers but found it quickly got very messy when accessing array contents and defining array capacity. Thus, we decided our implementation was too limiting of the functionality offered by Lua tables, so we abandoned this path feeling they were best left as higher-level structures not to be touched in our lower-level optimizations.

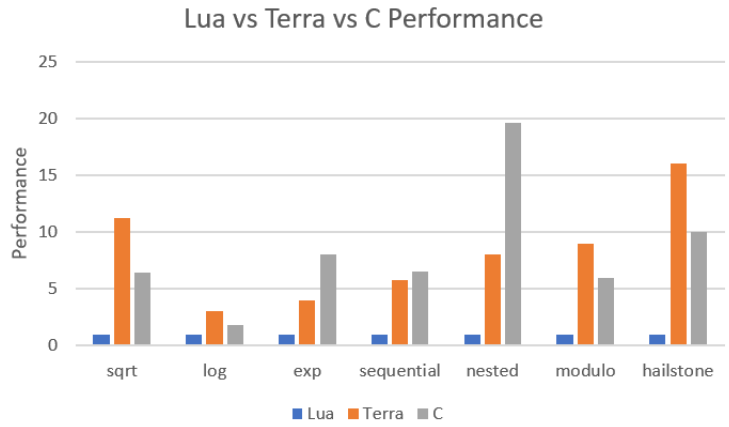
2.3 Performance Testing

We then generated a suite of test cases, ranging from basic arithmetic operations to more complex recursive calls such as Fibonacci and Hailstone sequences comparing the performance of each variation. Initially, we had intended to generate optimizations for simple binary operations, but our first round of testing showed that single Lua and Terra binary operations have very insignificant differences in performance; it is only when amplified thousand-fold do those differences emerge. Hence, our test cases moved to testing high-iteration loops and deeply-nested recursive calls.

Although in the suite of test cases we generated, optimized Terra functions could be compiled to an executable/object file since they do not actually have any Lua dependencies [2], in more realistic use cases, this would not be possible given the intended cooperation between Lua and Terra, so our performance tested speeds of interpreted Lua and JIT-compiled Terra.

3 RESULTS

For our data collection, we compared performances of corresponding Lua, Terra, and also C (at -Ofast optimization level) to see how well Terra serves its intended role of being a high performance low-level language. In the data below, we show the most significant of results: we first compared across various math library functions, situated within highly-iterative loops, then across sequential and nested math functions, a custom modulo method [7], and the recursive hailstone sequence function [5].



We see a significant increase in performance between regular Lua and our optimized Terra functions. One interesting trend of note, however, is that, in general, C tends to outperform Terra when it comes to standard math library function calls, but then falls behind as computation becomes more rigorous (beyond simple function calls), as with the modulo and hailstone tests.

The results between Lua and Terra were as expected - Terra far outperformed Lua. We once again see the benefits of compiled, specifically JIT-compiled, languages versus interpreted languages and the tradeoffs that result when balancing ease-of-use and performance - this was, after all, the goal of the entire Terra project [8]. Admittedly, there requires the initial overhead of running our module to generate the optimized Terra code, but beyond that, performance is far superior. However, the more interesting comparison lies between Terra and C. We believe that the reason C outperforms Terra in standard math library-related tests is that the C standard library has been around for 20+ years and has undergone numerous revisions for optimal performance [4]. However, Clang, the LLVM frontend, is a more recent invention, only being around 10 years old. Nevertheless, when it comes to more higher-level computation tests, Clang, which boasts faster compilation times, optimizations, and a lower memory footprint, surpasses gcc (C) performance [3]. Thus, while outperformed by C in more rudimentary computations, Terra interfaces far better with Lua in terms of interoperability, boasts stellar performance, and has the benefit of leveraging the still-developing LLVM. Because Terra seems to be such a versatile and robust language, perhaps in the future, further developments on our module, related to increased type recognition and functional targets, could really increase usage of this language.

4 CONTRIBUTIONS

I did all the work for this project.

REFERENCES

[1] 2017. About Lua. (Dec. 2017). https://www.lua.org/about.html
[2] 2017. About Terra. (Dec. 2017). http://terralang.org/
[3] 2017. Clang-LLVM. (Dec. 2017). https://clang.llvm.org/comparison.html
[4] 2017. GNU C Standard Library. (Dec. 2017). http://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_1.html
[5] 2017. Hailstone Sequence. (Dec. 2017). https://rosettacode.org/wiki/Hailstone_sequence
[6] 2017. Lua Parser. (Dec. 2017). https://github.com/andremm/lua-parser
[7] 2017. Terracuda. (Dec. 2017). http://willcrichton.net/terracuda/
[8] Zachary Devito. 2014. *TERRA: SIMPLIFYING HIGH-PERFORMANCE PROGRAMMING USING MULTI-STAGE PROGRAMMING.* Master's thesis. Stanford University.