Persistent Cache in Lua

Daniel Grazian

Stanford University
dgrazian@stanford.edu

December 14, 2017

Abstract

Caching is critical to avoid unnecessary repetition of computations. Languages such as Python have modules for persistent caching of function results. As far as I have been able to find, Lua has no such feature. In this paper, I describe an effective and easy-to-use caching system in Lua.

I. Introduction

Caching is done at both the software and hardware level to avoid repeating computations. At the software level, a cache can be in-memory or it can be persistent. An in-memory cache lasts for the lifetime of a program. The big advantage of an in-memory cache is that it is fast. A cache can be a simple hash table, enabling very fast access.

However, an in-memory cache has two significant disadvantages. First, it's limited by the size of main-memory. When caching very large amounts of data, this may not be enough memory. Second, it only persists for the lifetime of the program. If you execute the program again, all the cached values will have to be recomputed. This could be highly undesirable for applications involving expensive computations, such as database queries and network calls.

As a simple example, suppose you have a function that takes an input n and computes the sum of the integers from 1 to n. Clearly, if you expect the function to be called on the same number multiple times, it is advantageous to cache the results. We could imagine the following Python code (there are more imaginative optimizations, but this gets the point across):

```python
cache = {}
def sum_n(n):
    if n in cache:
        return cache[n]
    result = 0
    for i in xrange(1, n + 1):
        result += i
    cache[n] = result
    return result
```

The same idea is equally easy to write in Lua. This is better than nothing, but there are several improvements we would like to make:

- A cache should be encapsulated inside a function and not require global variables.
- Caching logic should be generic and not have to be repeated for each new function.
- Caching should be persistent.

Python makes this convenient with the use of decorators. A decorator is a reusable wrapper around a function that adds extra functionality to that function. For example, the Python code above could be better packaged as:

```
def memoized(f):
   cache = {}
   def decorated(*args):
      if args in cache:
         return cache[args]
      cache[args] = f(*args)
      return cache[args]
   return decorated

@memoized
def sum_n(n):
   result = 0
   for i in range (1, n):
      result += i
   return result
```

The *@memoized* decorator transforms *sum_n* into the function returned by applying *memoized* to *sum_n*. This neatly satisfies the first two points above: the cache is associated with one particular function, and the caching logic can be applied to any function with a simple annotation. Making caching persistent without sacrificing robustness is more involved (and the focus of this paper) but one can get an idea of the basics in Python at the blog post here.[1]. However, I am not aware of any generic persistent caching system of this sort in Lua.

This project is in keeping with the language-comparison theme prevalent in CS242. I am implementing an established Python feature in Lua and comparing the performance of the languages with respect to the feature. I also take extensive advantage of functions being first class objects in Lua.

II. Approach

As with Python's decorator functions, my approach is to write a Lua function that takes another Lua function as an argument and returns a function that performs caching logic on top of the first function. This is possible, because just like in Python, functions are first class objects in Lua. The following is the bare-bones outline:

```
function memoized(f)
  ...
  function decorated()
    ...
  end
  return decorated
end

function func1
  ...
end
```

[1]https://blog.jverkamp.com/2012/09/29/pickles-and-memoization/

```
func1 = memoized(func1)
```

Lua lacks the syntactic sugar of the *@decorator* syntax, but the same effect can be achieved by declaring $f = \text{memoized}(f)$ immediately after the declaration of a function f.

Now we need to consider what the *memoized* function should do.

We want to persist the cached values inside a directory. Each function must have its own cache directory, to keep its cache results separated from those of other functions. We don't want, say, $g(n)$ to fetch the cached result of a different function $f(n)$.

Each distinct input has its own file in the cache directory, containing the corresponding output. The file is named for the md5 hash of a serialized form of the function arguments. In addition to security benefits, the md5 hash is useful because it eliminates the possibility of reserved characters in the function name. The file contents are the serialized form of the function output for those arguments.

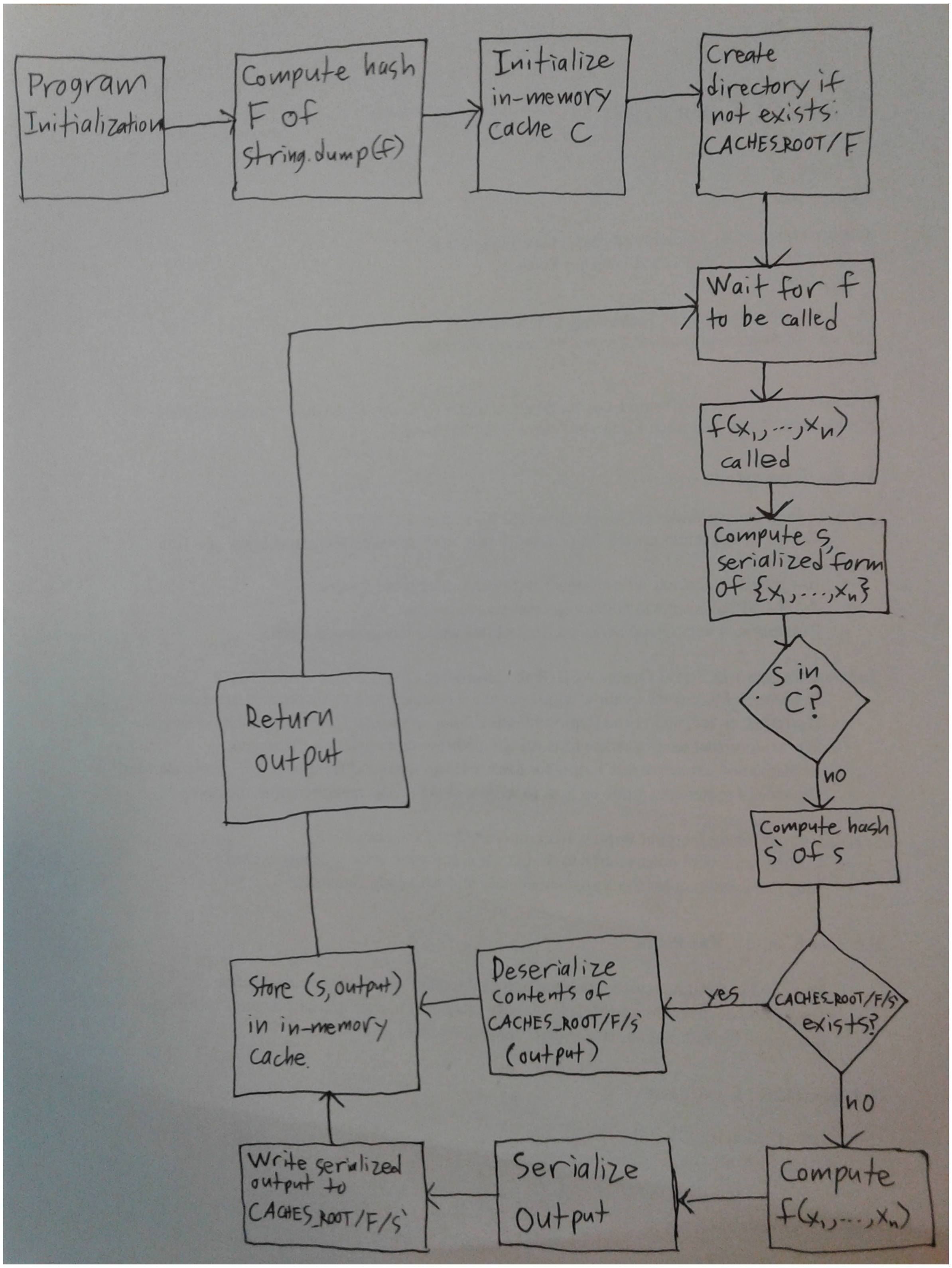
When a memoized function call $f(x_1, x_2, \ldots, x_n)$ is made, the cache directory for f is checked for a file name corresponding to $\{x_1, x_2, \ldots, x_n\}$ (again, this would be a hash of the serialized form of the arguments.) If the file exists, its contents are read, deserialized, and returned as the result. Otherwise, the function call is computed, the file is created, and the serialized output of the function is written to the file.

The serialization of the inputs and outputs to functions is a slightly more robust version of that from Assignment 1. An input of the form $(x_1, \ldots, x_n)$ is treated as the table $\{x_1, x_2, \ldots, x_n\}$ for serialization purposes. Tables are serializable (and thus supported as function inputs and outputs in this system) as are primitive values such as strings and numbers.

An important detail is that when the definition of a function changes between runs of a program, the cache results for that function must be invalidated. To achieve this, the cache directory of a function f is derived from a hash of string.dump(f). If the function changes in anyway, the value of string.dump(f) will change and a new directory will be created to hold the cache values.

My system contains an in-memory cache layered over the persistent cache. The motivation for this is that accessing data in-memory is much cheaper than accessing it on disk. When a function call is made, the in-memory cache is examined for a cached result; the persistent, file-based cache is only used if no result is found in the in-memory cache. Any function result, whether an original computation or found in the persistent cache, is stored in the in-memory cache.

An overall flow diagram of the system, *from the perspective of a memoized function* $f(x_1, \ldots, x_n)$ is shown below:


Program Initialization
Compute hash F of string.dump(f)
Initialize in-memory cache C
Create directory if not exists: CACHES_ROOT/F
Wait for f to be called
f(x1, ..., xn) called
Compute s, serialized form of {x1, ..., xn}
s in C?
no
Compute hash s` of s
CACHES_ROOT/F/s` exists?
yes
Deserialize contents of CACHES_ROOT/F/s` (output)
Store (s, output) in in-memory cache
Return output
no
Compute f(x1, ..., xn)
Serialize Output
Write serialized output to CACHES_ROOT/F/s`

I evaluated the performance of this module on some benchmarks. I timed its performance on computing large Fibonacci values, twice. I timed its performance on computing the sum of the first 1,000 triangle numbers, twice. In each case, I compared performance with and without the use of the persistent cache. I also compared performance with that of an equivalent module that I wrote in Python.

I also performed a large battery of correctness tests.

III. Results

All performance tests were done against a clean cache.

The following table shows performance on computing the 250th Fibonacci number:

	First run	Second run
Lua, memcache	104	0
Lua, no memcache	105	0
Python, memcache	453	0
Python, no memcache	476	0

Time to compute 250th Fibonacci number (ms)

Let's analyse the rows of the table one by one. Under the persistent cache framework we built for Lua, the first time Fib(250) is called, the computation takes approximately $100ms$. Computing the n_{th} Fibonacci number involves storing each smaller Fibonacci number in the (persistent) cache. The second time Fib(250) is called, the computation takes less than one millisecond. This is because the module can directly retrieve the result of Fib(250) from the persistent cache. Reading from a single short file takes much less than a millisecond. For this reason, there is little added value to using an in-memory cache in this case.

The performance benefits from caching are similar in Python. Lua is generically faster than Python, so it is gratifying to see that the Lua module is much faster than the Python module.

Not using any caching at all is disastrous. Brute force computation of the nth Fibonacci number is exponential in n.

It is worth noting that the computation of a relatively small Fibonacci number is a sufficiently inexpensive task that using a persistent cache in this way is generally overkill. Computing the $250th$ Fibonacci number from scratch (using standard "dynamic programming" memoization) takes less than a millisecond. Persistent caching is best suited for expensive computations, those whose cost is significantly greater than the cost of reading a value from a file. An example of which we are about to see.

The following table shows performance on adding the first 1000 triangle numbers. Note that the triangle(n) function, which computes the nth triangle number, is memoized, but the client function, which computes $\sum_{i=1}^{n} \text{Triangle}(n)$ is not memoized.

	First run	Second run
Lua, memcache	411	5
Lua, no memcache	454	36
Python, memcache	2022	0
Python, no memcache	1917	55

Time to compute the sum of first 1000 triangle numbers (ms)

We now see that there is a noticeable benefit from layering an in-memory cache on top of the persistent cache. In the second run, storing the triangle numbers in-memory saves the cost of opening and reading from 1000 files.

I also performed extensive correctness tests. I validated that the Lua persistent cache module is easily imported into other projects. I validated that the caching system works on functions that take large numbers of arguments or that accept deeply-nested tables. I validated that the module works fine when two unrelated files in the same directory have a memoized function of the same name (they do not interfere with each other's caches.)

Additionally, I had a friend use the persistent cache module, and she considered it to be convenient and easy-to-use.

IV. Conclusion and Future Work

I built an effective and easy-to-use generic persistent caching system in Lua. This system allows a programmer to essentially annotate a function and make it save its computations to disk. Additionally, the persistent caching system has an in-memory cache layered on top of it, saving IO costs where possible. The persistent cache system is most useful for very expensive computations, such as network calls or even just adding many numbers.

Although I definitely did test performance, I was mostly focused on functionality in this project. Future work could involve optimizing the cache to make it run as fast as possible (speed is the very purpose of the cache after all.) I would also be interested in conducting a usability study to see how widely such a feature would be among Lua programmers.

References

[Verkamp, John-Paul] https://blog.jverkamp.com/2012/09/29/pickles-and-memoization/