**Improving computational performance of function "determine\_optimal\_strategy\_v1"**

From the code you've provided, the main computational bottlenecks appear to be within the nested loops which process each strategy, and the operations within those loops that are performed for every possible starting point in time for the specified time horizon.

Here are some potential improvements you can make to boost performance:

1. **Profiling**:

First, it's crucial to identify where the most computational time is being spent in your code. You can use a tool like profvis in R to visually inspect the amount of time being spent in different parts of your function.

1. **Vectorized operations**:

In R, operations on vectors are faster than looping over individual elements. If there are any places where you can replace loop operations with vectorized operations, it could improve your function's performance.

R is a vectorized language, which means it can perform operations on entire vectors or matrices in a single operation rather than doing it element-by-element. Where possible, replace loops with vectorized computations.

Vectorized operations refer to operations that are performed on whole vectors or matrices in a single step, rather than on individual elements. This is generally more efficient in R. For example, instead of using a for loop to calculate cumulative returns, you could try to use cumprod function directly on the vector.

**2B FAMILY FUNCTIONS INSTEAD OF LOOPS?**

xyzxyzxyz

1. **Memory pre-allocation**:

In the loops you're using, the size of the object is increasing at each iteration (for example, df\_series\_to\_plot <- rbind(df\_series\_to\_plot, df)), causing the entire object to be copied and rewritten to memory at each step. Pre-allocating the size of objects when you know their final size will greatly speed up your function.

When you grow an object (like a vector, list, or data frame) inside a loop by adding elements or rows to it, R has to reallocate memory for that object at every iteration which can be time-consuming. It's more efficient to preallocate memory for such objects, if you know their final size.

Memory pre-allocation is a method of creating an object of the size you need before you start filling it with data, which can help to reduce computation time by reducing memory reallocations. For example, initializing df\_series\_to\_plot before the loop and then using list indexing to fill it could be faster.

1. **Parallel computation**:

If your computations are independent, as it seems to be in the main loop over each strategy, you can make use of parallel processing to compute results for multiple strategies simultaneously. Packages like parallel, foreach, future.apply in R can be used for this.

In the case where computations for different strategies are independent from each other, you could parallelize the code to make use of all cores in your processor. You can achieve this with the parallel package in R. However, please be aware that parallel computing comes with its own set of challenges, like setting up communication between cores and avoiding race conditions.

Parallel computation is a method to execute multiple computations at the same time. R has packages for this like parallel, foreach, and doParallel. Depending on the number of cores you have, you could break the for loop into chunks and execute them in parallel.

1. **Reduce number of operations**:

Your code is creating a plot for each strategy and for every year even though all the plots are not being returned by the function. This could be a potentially expensive operation. You might want to consider moving this out of the main calculation loop or consider ways to reduce the number of times the plotting occurs.

For example, cum\_returns\_time\_period[length(cum\_returns\_time\_period)] is computed twice in the code, once for checking lowest cumulative return and once for checking highest. It would be more efficient to compute it once and store it in a variable, then use that variable later.

Sometimes, fewer operations can be performed by writing more efficient code. For instance, in the loop, you calculate cum\_returns\_time\_period[length(cum\_returns\_time\_period)] twice. You could store this in a variable and reuse it, reducing the number of operations.

1. **Memory efficiency (Use more efficient data structures):**

Using more memory efficient data structures can speed up the execution of your function. For instance, instead of using data frames, you can use matrices or arrays when your data doesn't include different types.

The data.table package in R provides an interface to manipulate large datasets that is faster and more memory efficient than data.frame. Especially when you do operations like rbind, which can be quite slow on large data frames, data.table can provide significant speedup.

Certain data structures can be more efficient for certain tasks. For example, data.table is often faster than data.frame for large datasets.

1. **Use optimized libraries**:

Some R packages are optimized for fast operations on large data sets. For example, data.table is an enhanced version of data.frame that provides a high-performance version of base R's data.frame.

For example, for more efficient operations with dates, you could use the lubridate package. For dealing with missing data or more efficient operations on data frames or tibbles, tidyverse and dplyr can be very useful.

There are libraries in R that are optimized for speed. An example of this is the Rcpp package, which allows you to write C++ code in R.

**8. Memoization/Caching:**

For parts of your code where you're doing the same computation multiple times, you can use memoization or caching to store the result of a computation the first time it is done, then just look up the stored result the next time you need it, instead of recomputing it.

Memoization is a technique where you store the results of expensive function calls and return the cached result when the same inputs occur again.

**9. External computation:**

**If the dataset is extremely large, consider using an external computation resource such as cloud-based options (AWS, Google Cloud, etc.) or R's interface to databases which might handle large datasets more efficiently.**

# Profiling

*Once the function completes its execution, a new window will pop up in RStudio showing the profile. This will display a flame graph, where the y-axis represents the call stack depth and the x-axis represents the execution time. By hovering over different blocks in the graph, you can see which functions are taking the most time.*

*This can help you identify the bottlenecks in your code and where you might want to focus your optimization efforts.*

On the smaller dataset:

|  |  |
| --- | --- |
| **Part of Function** | **Execution Time (seconds)** |
| Part 1 | 0 |
| Part 2 | 9.1 |
| Part 3 | 0.12 |
| Part 4 | 0.11 |
| Part 5 | 7.44 (Sum of 20 runs) |
| Total Function | 9.33 |

**Interpretation**

Part 1 initializes variables and containers for data. Its execution time is negligible, meaning it's highly efficient.

**Part 2 is the most time-consuming, as it loops over each strategy, then for each strategy, it loops over all possible starting times. This double loop creates a time complexity of O(n^2), leading to increased execution time with larger input data.**

**Part 5 is a sub-part of part 2, specifically the inner loop. It is run 20 times based on the output provided and the total time spent is 7.44 seconds. The repetition of this loop increases the total execution time significantly.**

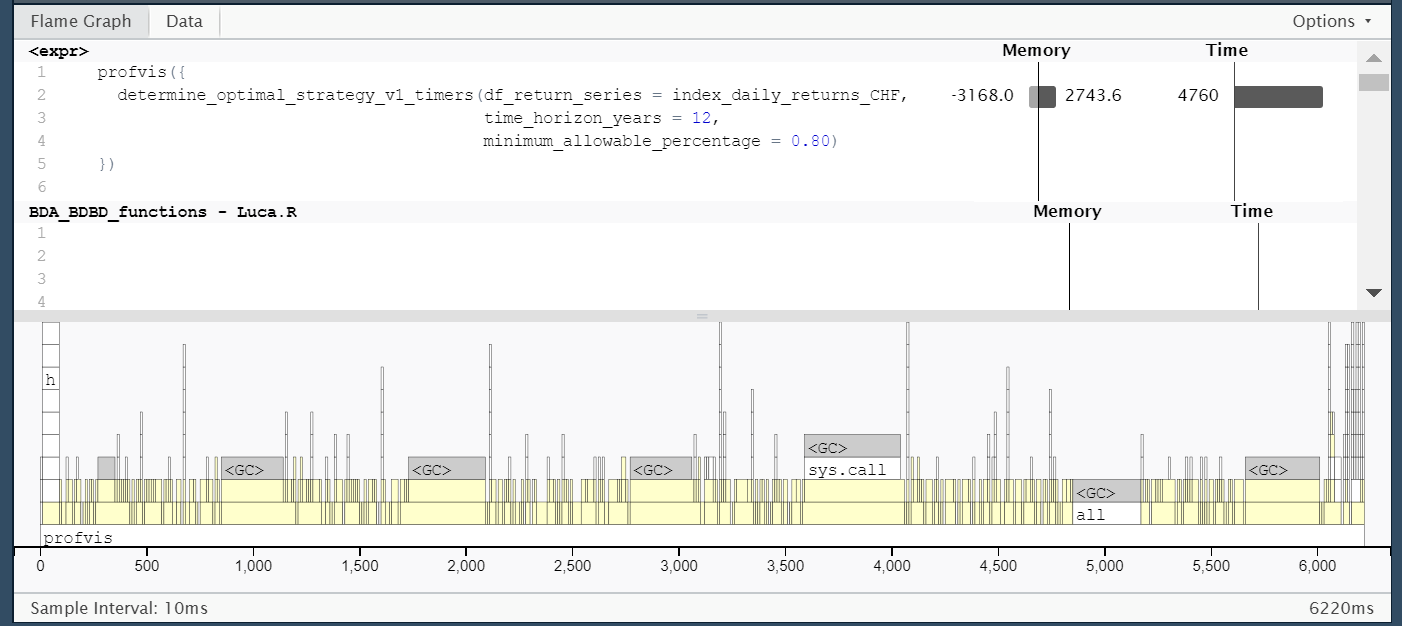
Part 3 selects the optimal strategy based on the results obtained in Part 2. The execution time is also negligible, suggesting its efficiency.

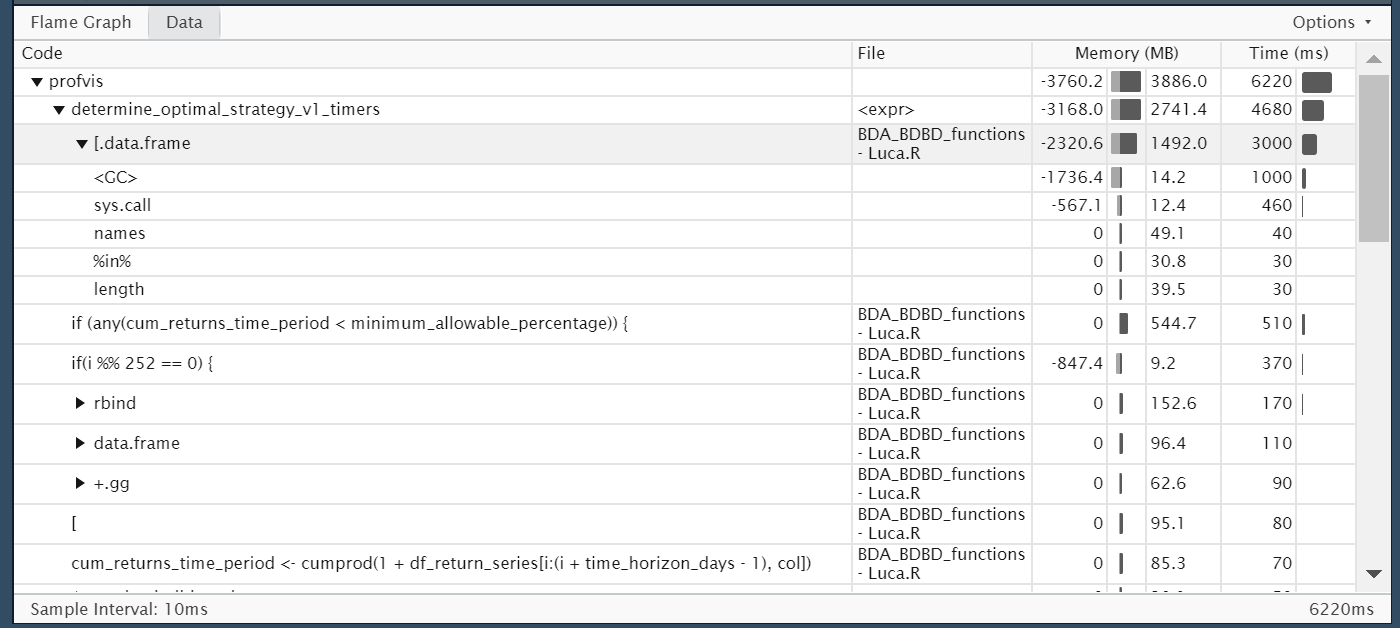
Part 4 generates a plot based on the data obtained. Although this task might seem heavy, its execution time is relatively short, suggesting efficient computation.

Suggestions for Improvement

1. **Part 2 and Part 5**: Given the major part of the execution time is spent in these parts due to looping, consider exploring methods to optimize this. Specifically, **see if it's possible to vectorize operations or use apply family functions instead of loops**.   
   --> **If the loops are unavoidable, consider parallelizing your code** using packages like foreach, parallel, or multicore if your hardware supports it.

Still on the smaller dataset:





On the larger dataset:

