**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.

The profvis package in R allows you to visualize where your code is spending the most time. By profiling your code, you can identify which parts of your code are the most time-consuming and therefore the best candidates for optimization.

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.

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.

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.

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.

1. **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.

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.

**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.