A Model for The Growth Factor In Allocated Memory of Golang's Slices

Mariana Ávalos Arce
Wispok Research
mariana.avalos@wispok.com

Abstract

The purpose of this document is to analyze a use case program where Golang's slices are large enough to compromise the memory usage of the machine running the corresponding program, that is, near the values of 2 GB of RAM memory. Such case is analyzed with the aim of proposing an equation describing slices' behaviour, as well as an explanation to the memory usage throughout the various versions of the aforementioned program, each of them optimized in a different manner.

1 How A Slice Works

1.1 The `make` Function

Golang's `make` function allocates a new array in memory and creates a slice header to point at it. Said function takes three parameters, crucial for further reading: data type, the initial length (assuming one will perform an `append` on it) and the capacity, which is the length that the allocated array has. If the third argument is omitted, the capacity is set to be equal to the length.

1.2 The `append` Function

After we create a slice using `make`, since the slice data structure has the purpose of dynamic resizing, it is common to perform several `append` operations on the slice in order to add new elements at the end of the underlying array. However, in order to give the user the appending behavior she wishes, Golang performs several processes under the hood. Whenever a slice is subject to an `append` call, Golang checks whether the underlying array to which the slice points to has the *capacity* to add such new elements, if said capacity C is not enough, then the method `growslice` is called to compute the new capacity C', creates a new array with length C', copies the contents of the old array to this new, larger array and then adds the new elements at the end of said new structure, to finally return the pointer to the user.

This clever approach may compromise usage of volatile memory, especially in machines of limited resources, since at some point during runtime two arrays coexist instead of one: when the old, insufficient array needs to be copied to a new, larger one in order to add new elements, two very similar structures are allocated in Random Access Memory while `copy` is carried out. It is then important for the programmer to understand how Golang performs the slice growth process, and the beginning of this understanding is to know how the new capacity C' is calculated.

1.3 The `growslice` Method

A snippet of the implementation of the slice structure's `growslice` method reads:

```
newcap := oldCap
doublecap := newcap + newcap
if newLen > doublecap {
    newcap = newLen
} else {
    const threshold = 256
    if oldCap < threshold {
        newcap = doublecap
    } else {
        // Check 0 < newcap to detect
            overflow
        // and prevent an infinite loop.
        for 0 < newcap && newcap < newLen {
            // Transition from growing 2x
                for small slices
            // to growing 1.25x for large
                slices. This formula
            // gives a smooth-ish
                transition between the two.
            newcap += (newcap + 3*threshold
                ) / 4
        }
        // Set newcap to the requested cap
            when
        // the newcap calculation
            overflowed.
        if newcap <= 0 {
            newcap = newLen
        }
    }
}
```

The code snippet above describes the way in which the new capacity C' is calculated. Let s be a slice with N elements (length) and capacity C. If we want to append X elements to s, what will happen according to the source code is:

- A possible double length is stored as $2C$.
- If $X + N > 2C$, then the new capacity $C' = X + N$.
- Otherwise, if $C < 256$, then the new capacity is set to $2C$.
- For slices with a capacity (underlying array length) $C \geq 256$, the algorithm computes the new capacity C' in a loop where each iteration a value of $\frac{C+3(256)}{4}$ is accumulated in the C' variable, until $C' \geq X + N$.

The last statement is accompanied by a commentary where it is stated that such instruction is intended to make a *smooth* transition between a factor of 2 for small slices (underlying length smaller than 256) and a factor of 1.25 for large slices (underlying length greater or equal to 256). How this instruction actually makes a *smooth* transition will be studied in the next section.

2 A Slice's Growth Factor Function

The capacity of a slice reaches a point where it needs to increase. The form in which Golang grows the slice capacity x by a certain amount a can be expressed as:

$$a = \frac{x + 3(256)}{4}, \tag{1}$$

where the growth factor of the new capacity x after the addition of a can be 2.0 for *small* slices, which Golang sets as those with less length than 256, all the way until a factor of 1.25, for any slice with length equal or larger than 256. We can thus define a function that describes the behaviour of the growth factor given a and by defining the factor as y:

$$y = \frac{x + \frac{x+3(256)}{4}}{x} \tag{2}$$

If we further simplify the function,

$$\begin{aligned} y &= \frac{x + \frac{1}{4}\left(x + 3(256)\right)}{x} \\ &= 1 + \frac{\frac{x + 3(256)}{4}}{x} \\ &= 1 + \frac{x + 3(256)}{4x} \\ &= 1 + \frac{x}{4x} + \frac{3(256)}{4x} \\ &= \frac{5}{4} + \frac{3(256)}{4x} \end{aligned} \tag{3}$$

we reach a simplified form that can be decomposed to its elementary transformations of the function $f(x) = \frac{1}{x}$ by writing:

$$y = \frac{5}{4} + \frac{3}{4}(256)\left(\frac{1}{x}\right), \tag{4}$$

which can be generalized as:

$$y = \alpha + \beta\gamma\left(\frac{1}{x}\right) \tag{5}$$

The equation in (5) with the constant values $\alpha = 1.25, \beta = 0.75$ and $\gamma = 256$ shows the following behaviour, where $256 \leq x < \infty$:

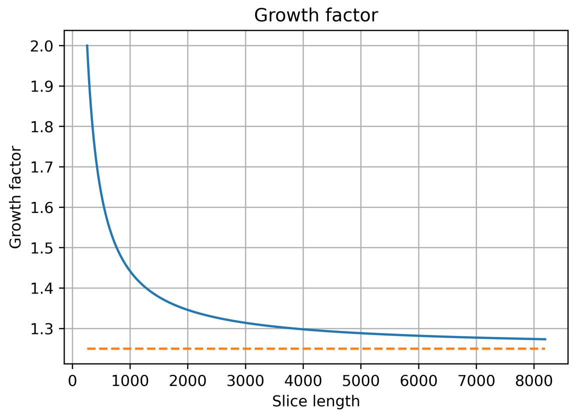


Figure 1: Growth factor plot.

3 Function Breakdown

If we seek to reach an explanation of every constant involved in the function

$$y = \alpha + \beta\gamma\left(\frac{1}{x}\right) \tag{6}$$
$$\text{where } \alpha = 1.25, \beta = 0.75, \gamma = 256,$$

and the function form $\frac{1}{x}$, then let the only known data to be the maximum factor for *small* slices, that is $\lambda = 2.0$, the minimum possible factor as the slice length $x \to \infty$, that is, $\alpha = 1.25$, and the threshold of length $\gamma = 256$. These three known values give us three major conditions to fulfill with the function in question:

- A function with an asymptote on line $y = \alpha$ as $x \to \infty$.
- A function that determines the output factor with respect to 256 as $x \to \infty$, which follows the form $\frac{256}{x}$.
- One of the points reached by the function is $(256, 2)$.

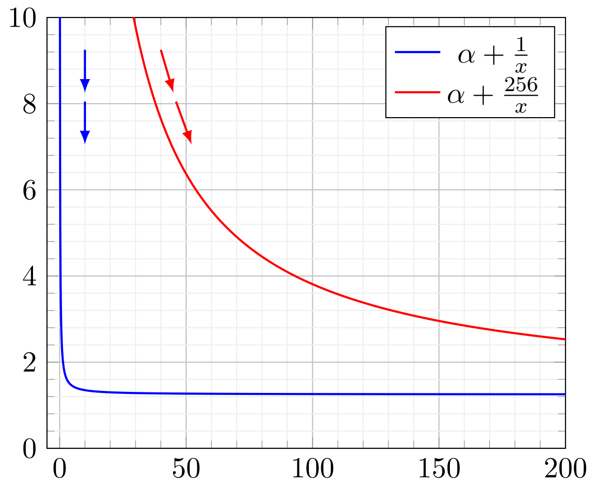
Having said that, we know that as the slice length grows from 256 towards infinity, the factor approaches **but never goes further** α. Such statement is the definition of a function with a horizontal asymptote on the line $y = \alpha$. From that, we now have two possible intuitive approaches for building the function:

$$y = \frac{1}{x - \alpha} \tag{7}$$

or

$$y = \alpha + \frac{1}{x}, \tag{8}$$

which both have the same horizontal asymptote on the line $y = \alpha$. Nevertheless, we must choose Eq. (2) form to continue, if we consider the third constraint or known value $\gamma = 256$: by having the value γ separate what we will consider a *small* or *large* length, the factor y will then be shaped by γ. In other words, the factor y must coincide with the **the amount of times length x fits in** γ, or γ divided by x. Otherwise, the function shape would follow the form $y = \alpha + \frac{1}{x}$, and by making it to follow the form $y = \alpha\frac{\gamma}{x}$, we are making the transition from λ to α use less steep slopes, and thus, the time in x to reach α from λ increases, as shown in Figure .



Therefore, if we plug in γ inside our two possible approaches' terms divided by x, we get:

$$y = \frac{\gamma}{x - \alpha} \tag{9}$$

or

$$y = \alpha + \frac{\gamma}{x}, \tag{10}$$

From these two probable approaches, we now can identify that Eq. (9) would indeed fulfill the condition of the asymptote on $y = \alpha$, but would not satisfy the condition of the factor being determined by the amount of times length x fits in $y = \gamma$ or $\frac{\gamma}{x}$, since Eq. (9) describes a factor determined by the amount of times x minus α would fit inside γ

$(\frac{\gamma}{x-\alpha})$. Therefore, the approach that fulfills both the asymptotic condition and the factor being determined by $\frac{\gamma}{x}$ would be Eq. (10).

Now that we have proceeded to build our function following Eq. (10), the only missing constant to conclude is β in order to achieve Eq. (5) intuitively from the known data. For this purpose, let us go back to our constraints: from having λ as the maximum possible factor when length $x = \gamma$, we have also one certain point to be reached by the function we seek to build, that is, the coordinates (γ, λ) or $(256, 2)$. Having said that, we have the values $y = 2$ and $x = 256$ standing as known. From this, we can plug in values $\alpha = 1.25$, $\gamma = 256$ and $x = 256$ in the function we have built so far:

$$\begin{aligned} f &= \alpha + \gamma \left(\frac{1}{x}\right) \\ 2.25 &= 1.25 + 256 \left(\frac{1}{256}\right) \end{aligned} \tag{11}$$

which means

$$\begin{aligned} 2.25 &\neq 2.0 \\ f &\neq y \end{aligned} \tag{12}$$

In Eq. (12), we conclude that the function f we have built up until this point gives us the coordinates $(256, 2.25)$ instead of the desired $(256, 2)$. Therefore, some factor β is missing inside the function we got so that the condition of having the coordinates $(256, 2)$ belong to our final function is fulfilled. The intuitive approach might be to add this missing value β to f, reaching

$$f = \alpha + \gamma \left(\frac{1}{x}\right) + \beta, \tag{13}$$

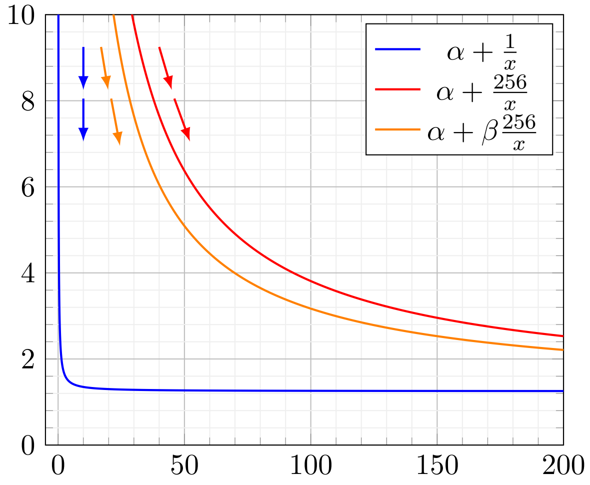
but the addition of β to function f would mean to move the asymptote to the horizontal line $y = \alpha + \beta$, and with this, f would not be satisfying the condition of having a horizontal asymptote on line $y = \alpha$. Therefore, β must not be included in f by addition. That leaves us with the aim of including β in f without moving the asymptote. One of the possible transformations that does not move the asymptote we have so far but indeed changes the output value of f as we desire is the multiplication of the term $\frac{\gamma}{x}$ by the required β. In this way, we reach the form

$$f = \alpha + \beta\gamma \left(\frac{1}{x}\right), \tag{14}$$

which includes the missing transformation of f by a value of β so as to fulfill the condition of having $(256, 2)$ belong to our final function, allowing us to solve for β if we return to our known values of $\alpha = 1.25$, $\gamma = 256$, $x = 256$ and $y = 2$, and plug them in Eq. (14):

$$\begin{aligned} y &= \alpha + \beta\gamma \left(\frac{1}{x}\right) \\ 2 &= 1.25 + \beta(256) \left(\frac{1}{256}\right) \\ 2 &= 1.25 + \beta(1) \\ \beta &= 2 - 1.25 \\ \beta &= 0.75 \end{aligned} \tag{15}$$

This involvement of β in Eq. (14) further transforms the version of the equation in Eq. (10), by making the slope in each point of the function in Eq. (14) to be steeper than the version in Eq. (10), but less steep than one of the early versions seen in Eq. (8). Figure shows such transitions.



Thus, we have a function that fulfills our initial conditions:

- Asymptote on line $y = \alpha$ as $x \to \infty$.
- A function that determines the output factor with respect to 256 as $x \to \infty$, which follows the form $\frac{256}{x}$.
- One of the points reached by the function is $(256, 2)$.

And said function is:

$$y = \alpha + \beta\gamma\left(\frac{1}{x}\right) \tag{16}$$

where $\alpha = 1.25, \beta = 0.75, \gamma = 256,$

which coincides with Eq. (3) that follows Golang's documentation. We can now conclude that the *smooth* transition from a factor of 2 for slices of length 256 towards a factor of 1.25 for slices of length ∞ can be built intuitively by stating the three conditions set for the factor's behaviour.

4 Results and Discussion

The motivation for the present work was to find a model to predict how Random Access Memory is being allocated so that a specific program was improved in order to avoid reaching memory limits in a machine with 2GB of RAM. The program's purpose is divided in two: first, is to cut large files (around 1 GB) into P pieces with `.cylf` file extension specific for this program; and second, to merge back all files with the aforementioned extension `.cylf` in order to reach the original file that was initially split in P parts.

Four versions of the mentioned program were developed, all reaching the same merged result but following different pipelines, with the purpose of visualizing how Golang's `growslice` method affects the machine's resources. Such differences in the `merge` method are described in the following table.

Version	`merge` pipeline
v1.0.0	For each f_i, append bytes one by one to slice of undefined size.
v1.0.1	For each f_i, append bytes as a whole to slice of undefined size.
v1.1.1	For each f_i, append bytes as a whole to slice of total size.
v1.1.2	For each f_i, write bytes as a whole to end of file in disk.

To further clarify, by declaring a slice of undefined size (i. e., with the line `var curr []byte`), the underlying array is equal to `nil` and the length is `0`. The case of $C =$ `nil` would fall directly in the case of $X + N > 2C$ and thus sets the new capacity to $C' = X + N$ in the first call to `append`.

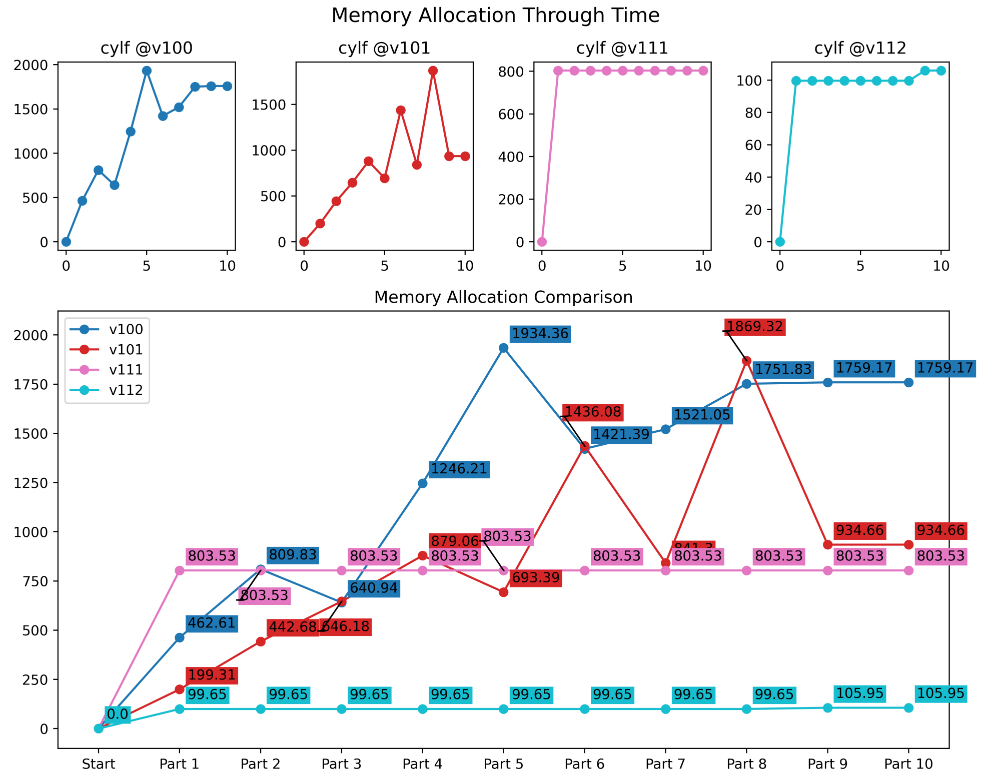


Figure 2: Memory allocation comparison between the four proposed pipelines.

A file with size of 766.7MB was used as input for the `cut` method, producing a total of 9 parts of size 95MiB, or 99.65 MB, which became the input for the `merge` method. As hinted before, in the first two versions of the program, the undefined size of the slice would fall into the case where $X + N > 2C$ and thus the new capacity after the first call to `append` would be $C = X + N$. However, since both length and capacity in the context of the current program is given in *bytes*, then after the first iteration the capacity would exceed Golang's built-in threshold of 256 that defines a *small* slice, and therefore, would in turn cause the iterative calculation of the new capacity for large slices. This behaviour is shown in Figure 3 at the leftmost plot, where after the first iteration, the length and capacity of the accumulating slice is 99 and 123, respectively.

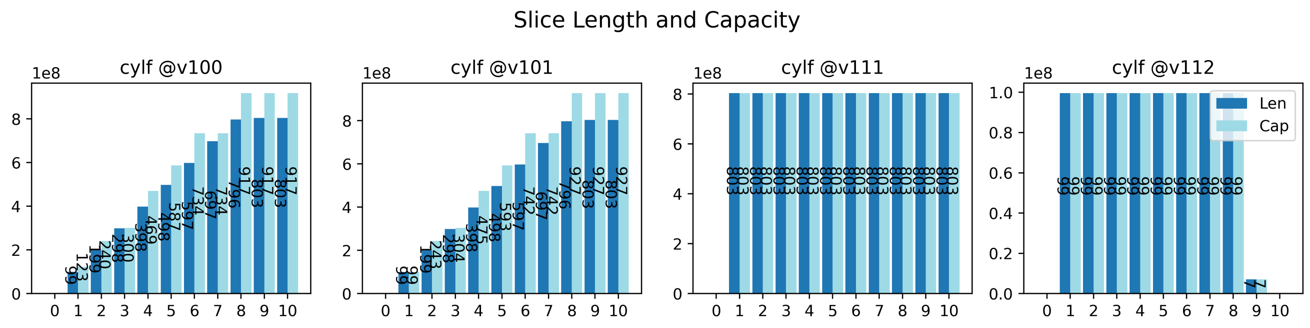


Figure 3: Length and capacity comparison between the four proposed pipelines.

Figure 2 showcases the behaviour of memory allocation throughout the different pipelines. If we compare memory allocation of version v1.0.0 and v1.0.1, where both pipelines append to an initially-undefined slice, we can see that the amount by which the programmer appends to the cumulative slice defines the future length and capacity of the slice as it is implied by Golang's documentation and code: in version v1.0.0, where we append byte by byte to a slice presents irregular jumps in memory allocation, whereas in version v1.0.1 the jumps begin to reach a constant rate in memory allocation increase since the appends are larger chunks of bytes. By comparing now versions v1.1.1 and v1.1.2, we can see that since the slice has the length and capacity set to $P \times f_i.bytes$, the underlying array never requires a resizing, and thus the RAM memory that the program requires in total can be anticipated as the size of the original file. This, although useful, will invariably make a machine stop the execution of the program in all cases where the machine has less RAM memory than the size of the original file. In turn, the program can seem unpredictable to the user: with some files works successfully and with others it will not resume execution. In order to avoid confusion in the user, version v1.1.2 was developed: slices are no longer required, since every file part gets written at the end of a file in the hard disk. In this way, the overall RAM memory usage required by the program is equal to the bytes required for one file part. Now, when using a machine with limited resources, the user will only be required to split the original file in as much file parts as possible, so that the RAM memory required to merge it back is smaller the more parts the file gets split into.

5 Conclusion and Future Work

This paper aimed to investigate the behaviour of Golang's slices when the method `append` is used, with the general purpose of providing a thorough analysis of the memory allocation required when working with slices, so that the programmer can be aware of the implications of the pipeline that she or he is implementing, especially when working with machines with limited resources in Random Access Memory (RAM). The outcome of the present work confirms that Golang's documentation and source code allows the programmer to anticipate and predict her or his programs' resource management.

The work presented here can be the basis for further study in analytical function transformations and its uses inside the field of Computer Science. Furthermore, other studies can be performed to explain with even more precision the behaviour of a program's memory allocation when working with Golang's slices or other data structure.

References

[1] A. Gerrand. *Go Slices: usage and internals*. URL: `https://go.dev/blog/slices-intro`.

[2] The Go Progamming Language. *slice.go*. URL: `https://github.com/golang/go/blob/master/src/runtime/slice.go`.

[3] R. Pike. *Arrays, slices (and strings): The mechanics of 'append'*. URL: `https://go.dev/blog/slices`.

[4] R. Surwase. *A Comprehensive Guide to Slices in Golang*. URL: `https://codeburst.io/a-comprehensive-guide-to-slices-in-golang-bacebfe46669`.