Control Theory and Concurrent Garbage Collection: A Deep Dive Into The Go GC Pacer

Madhav Jivrajani

`$ whoami`

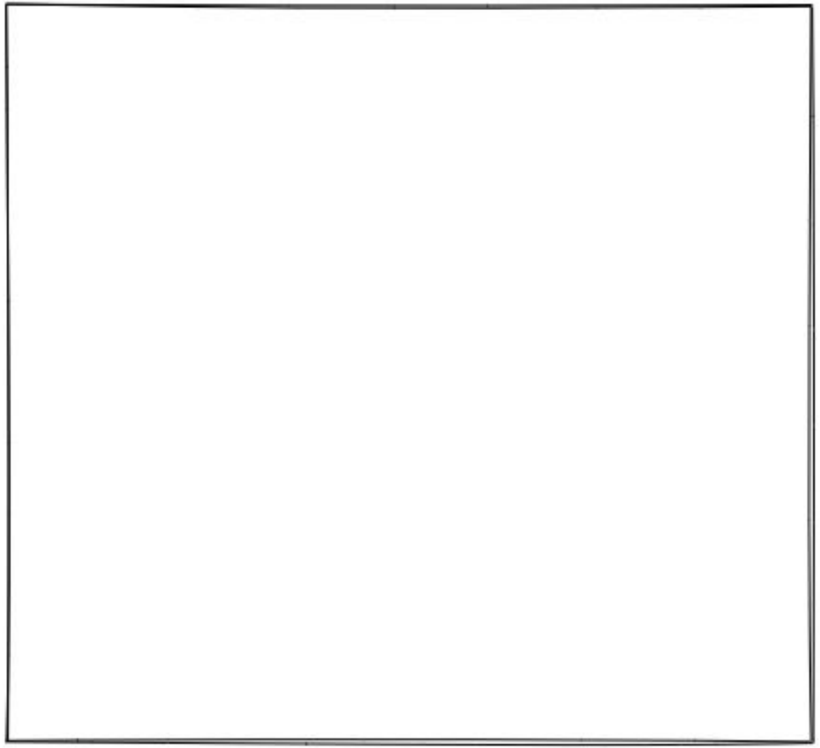
- I get super excited about systems-y stuff
- Work @ VMware (Kubernetes Upstream)
- Within the Kubernetes community - SIG-{API Machinery, Scalability, Architecture, ContribEx}.
    - Please reach out if you'd like to get started in the community!
- Doing Go stuff for ~3 years, particularly love things around the Go runtime!

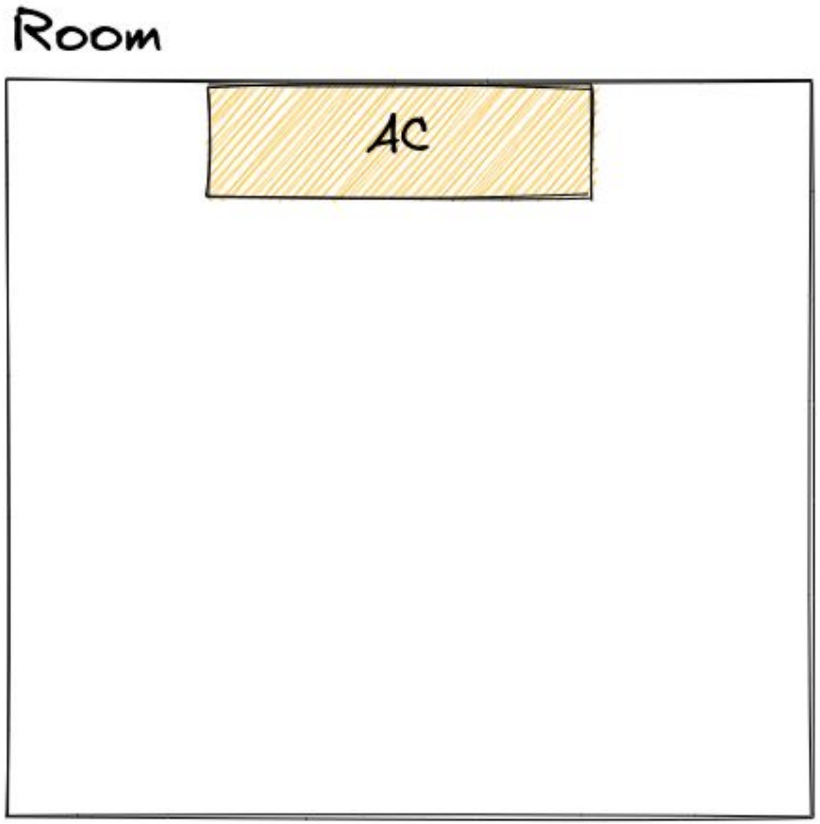
Agenda

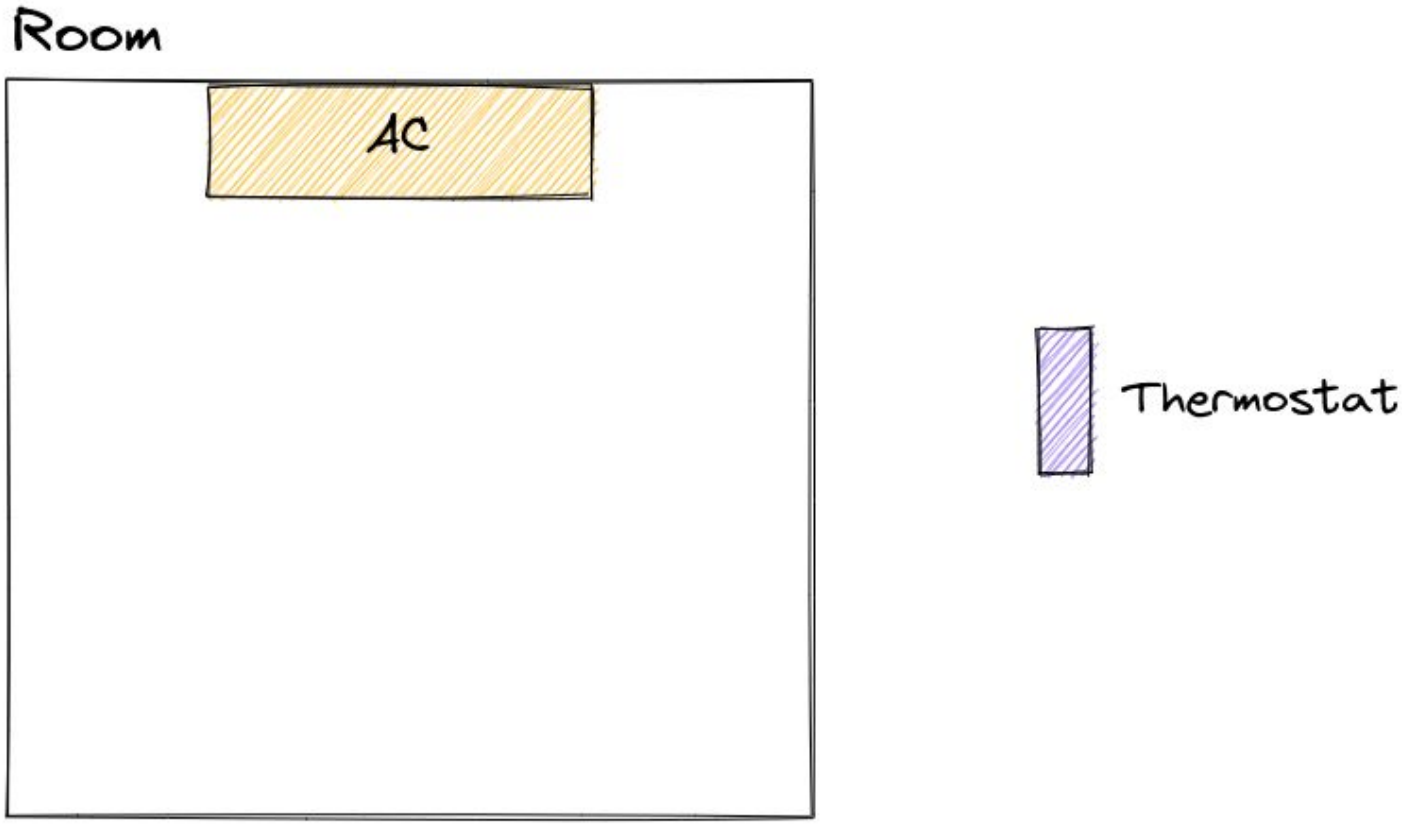
- Control Theory
- A Typical Go Application
- The GC Pacer
- GC Pacer Prior to Go 1.18
- GC Pacer Since Go 1.18
- How Did This Affect A Kubernetes Release?
- Mitigating These Effects
- Small Note On Go 1.19 Soft Memory Limit

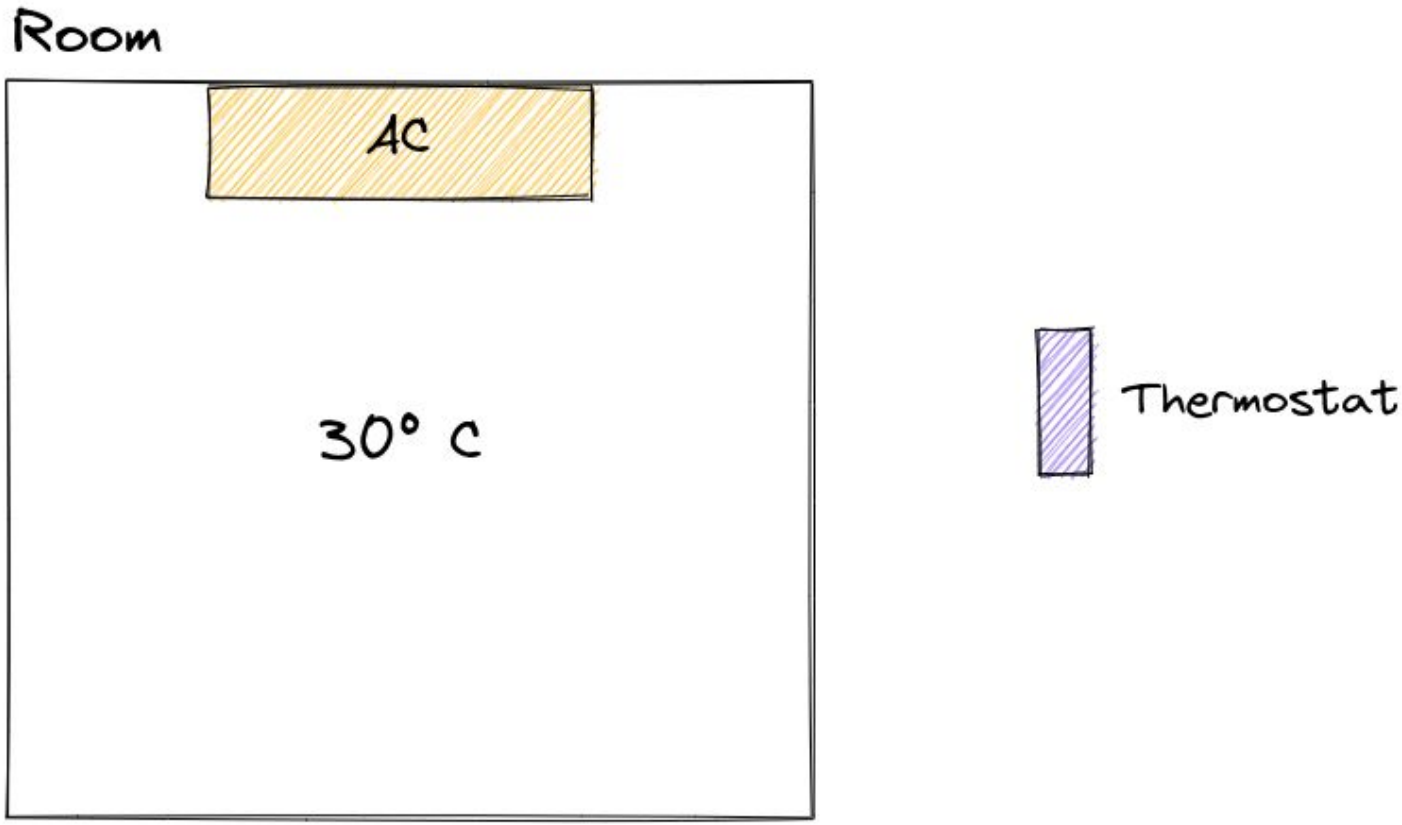

Control Theory

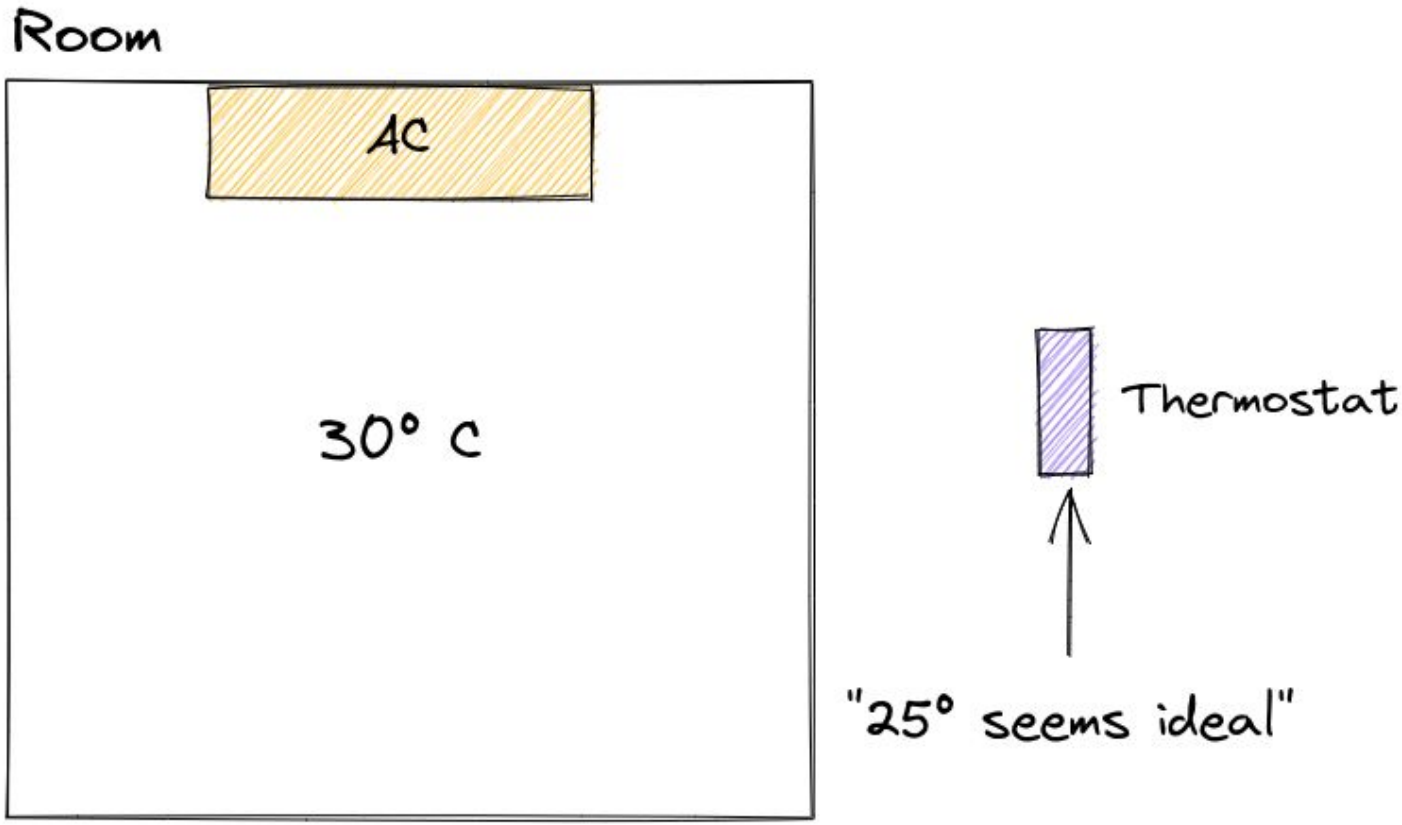
Room

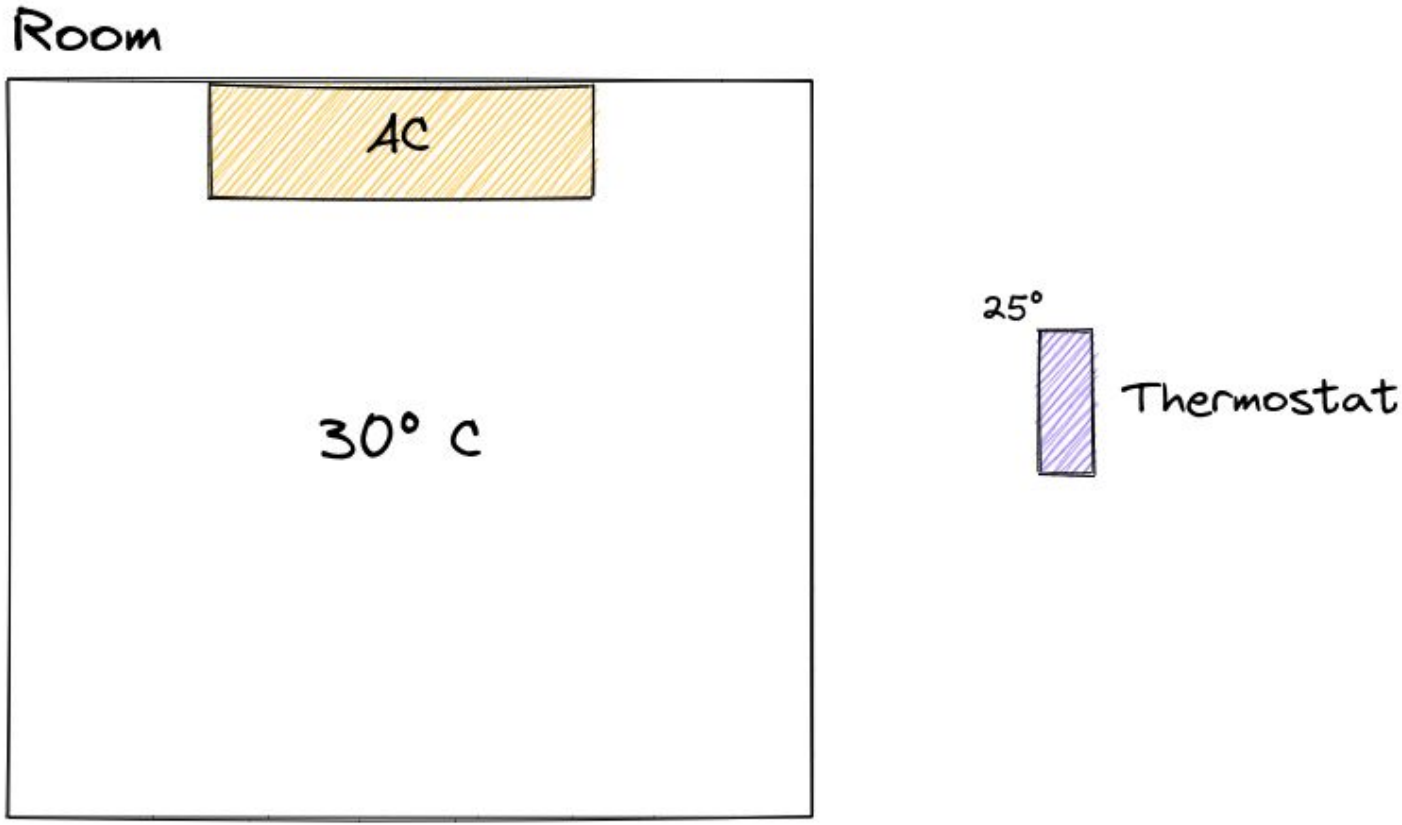


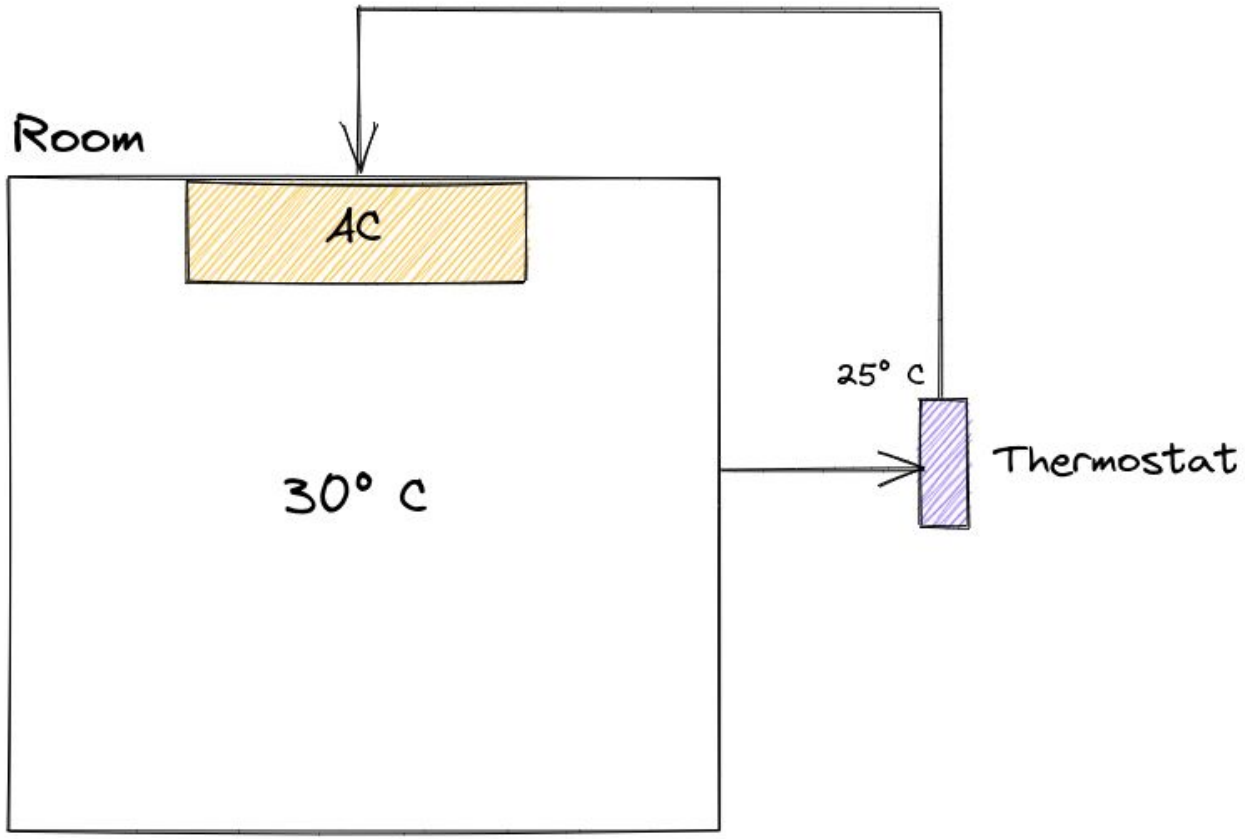

Room
AC

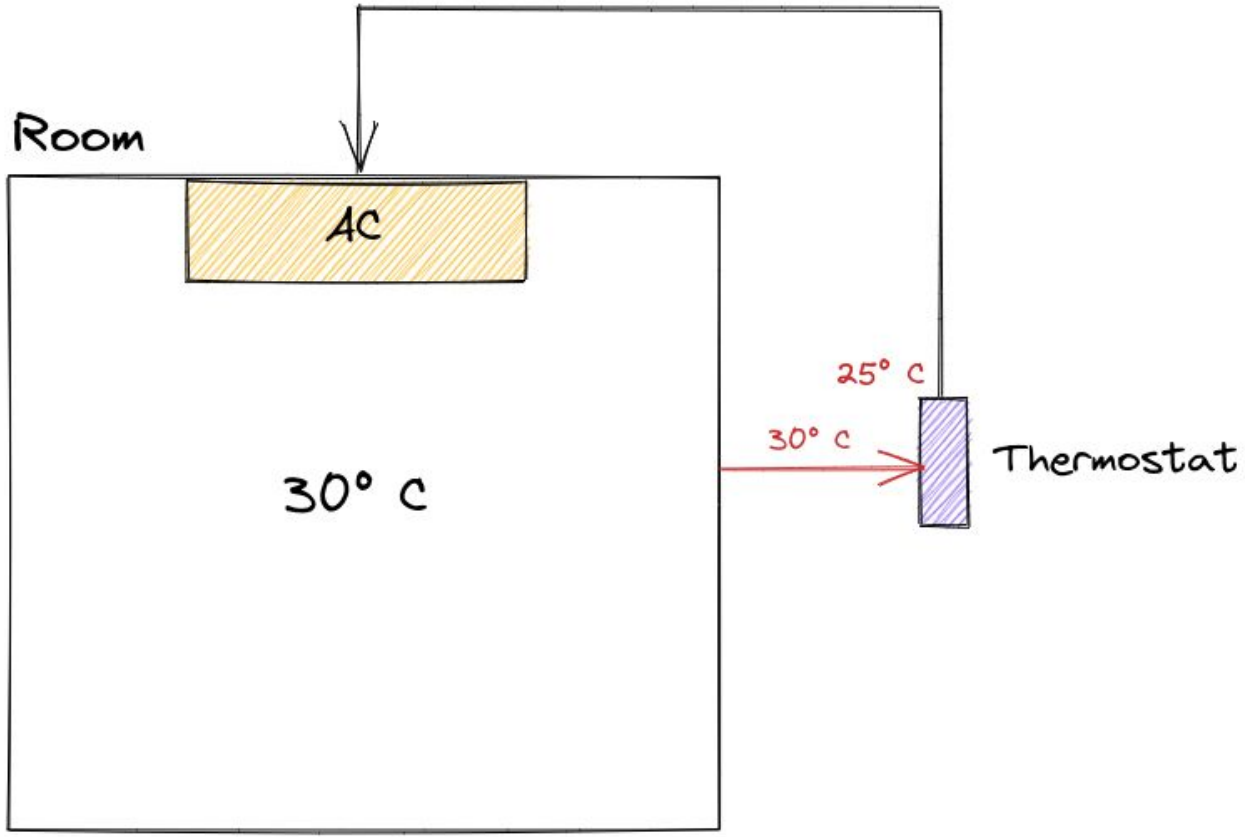

Room
AC
Thermostat

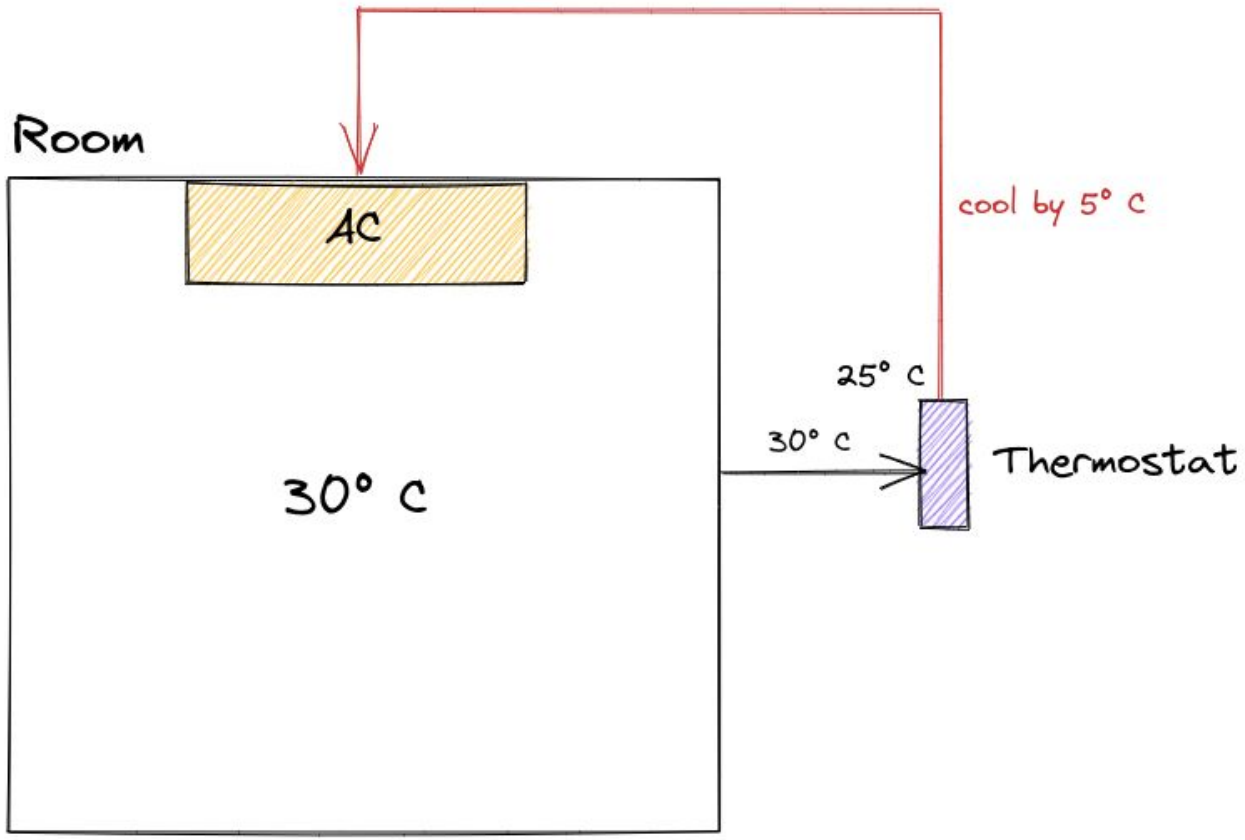

Room
AC
30° C
Thermostat

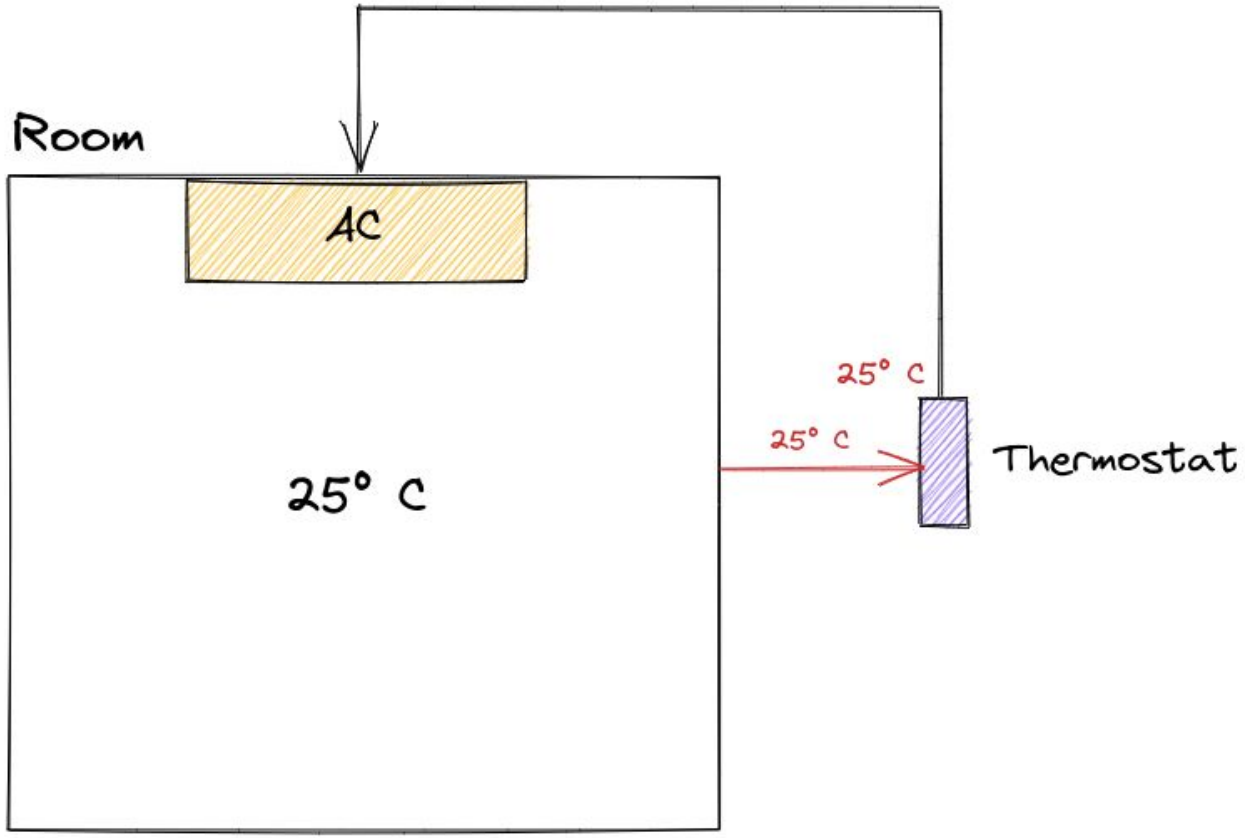

Room
AC
30° C
Thermostat
"25° seems ideal"

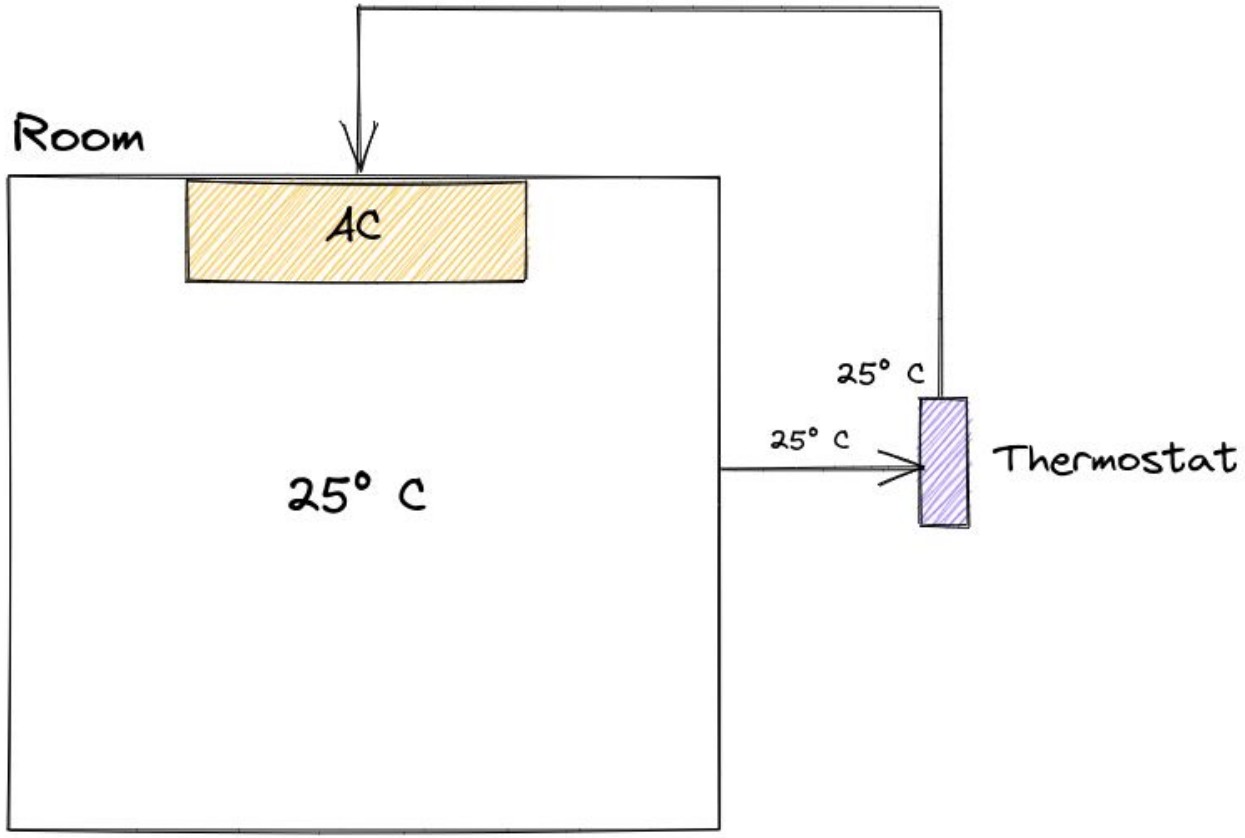

Room
AC
30° C
25°
Thermostat

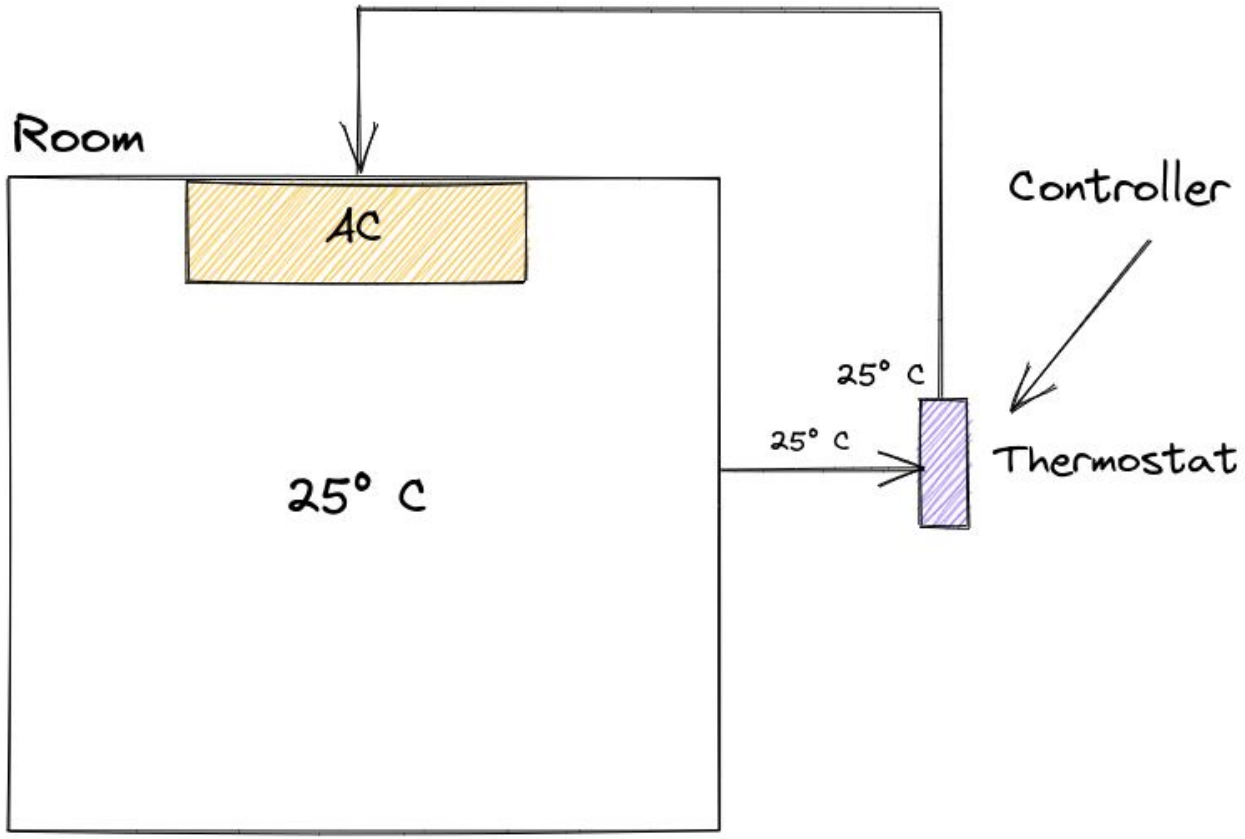

Room
AC
30° C
25° C
Thermostat


Room
AC
30° C
25° C
30° C
Thermostat

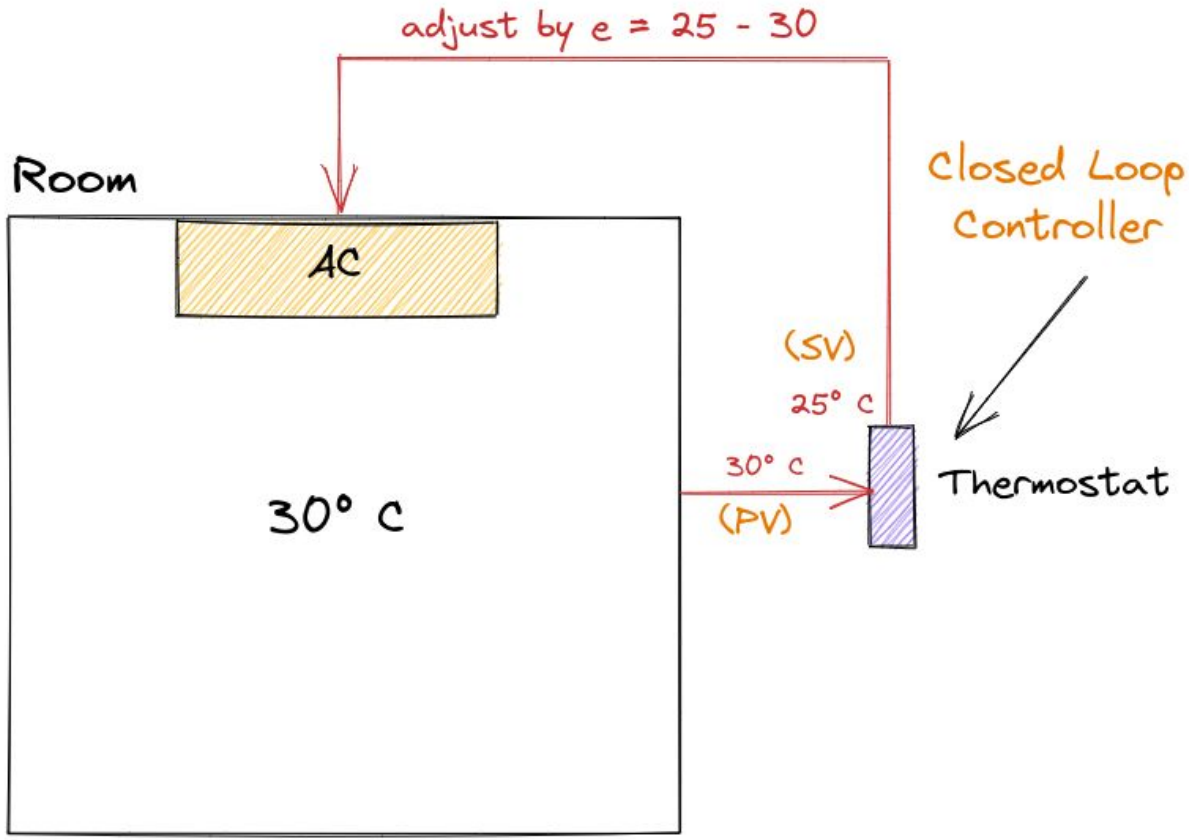

Room
AC
30° C
cool by 5° C
25° C
30° C
Thermostat


Room
AC
25° C
25° C
25° C
Thermostat


Room
AC
25° C
25° C
25° C
Thermostat

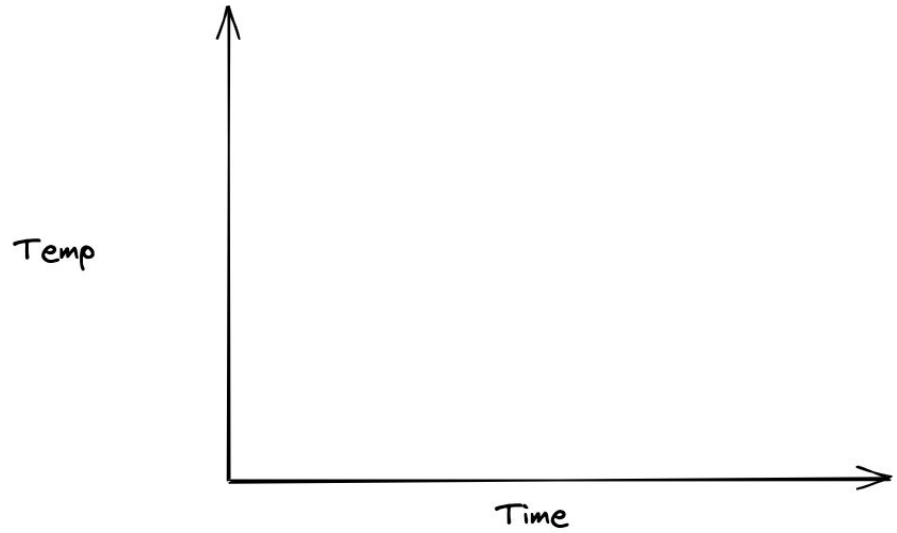

Room
AC
25° C
25° C
25° C
Controller
Thermostat

- SV - Set Variable (what we hope to achieve)
- PV - Process Variable (output of the system)
- Error - Difference between SV and PV

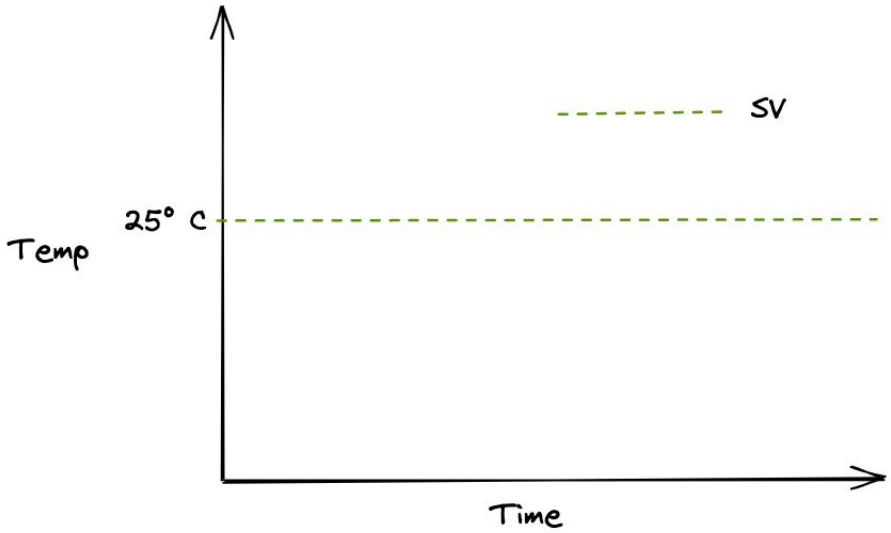

adjust by e = 25 - 30
Room
AC
30° C
(SV)
25° C
30° C
(PV)
Thermostat
Closed Loop
Controller

Transient and Steady State

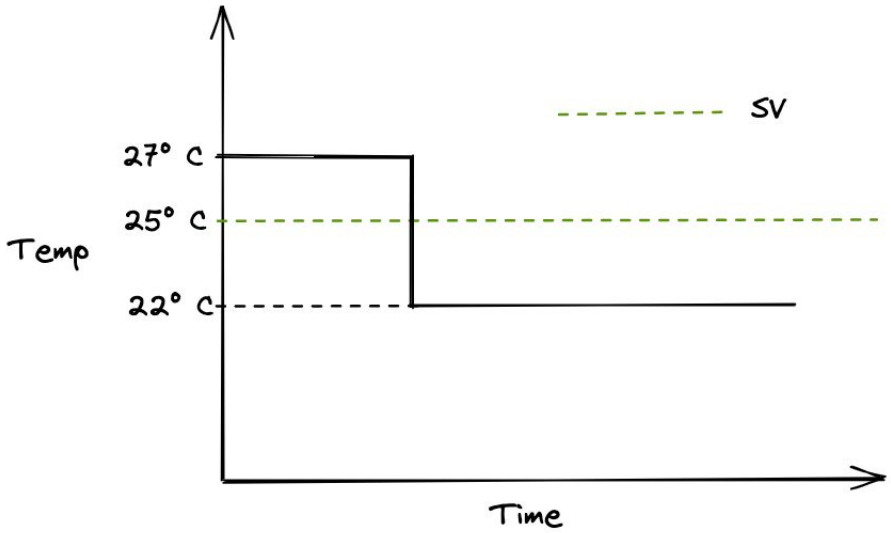
Transient and Steady State



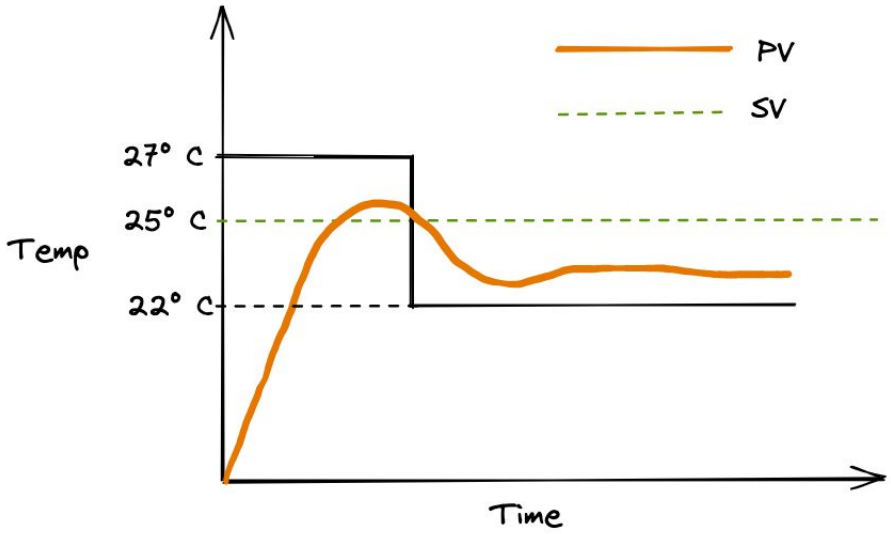
Transient and Steady State



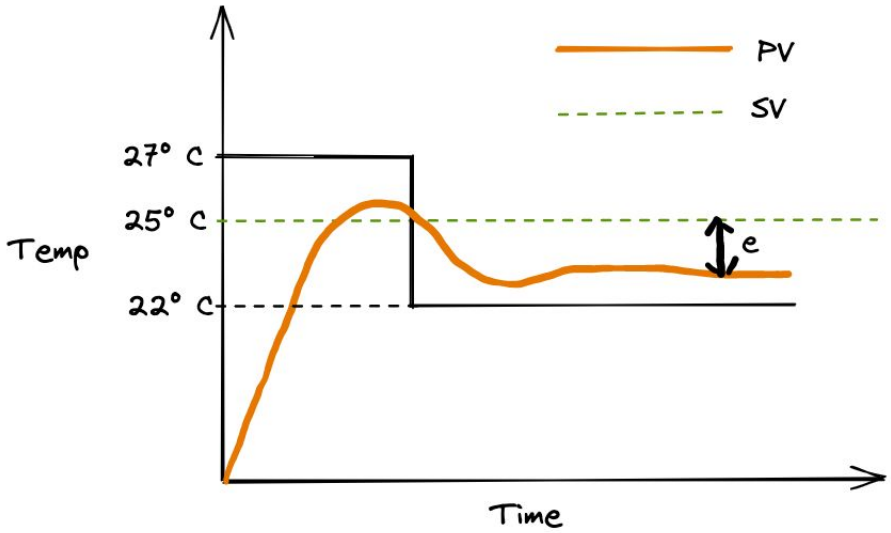
Transient and Steady State



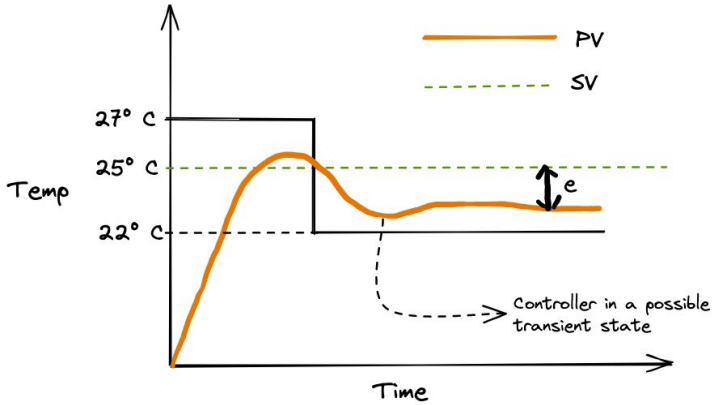
Transient and Steady State



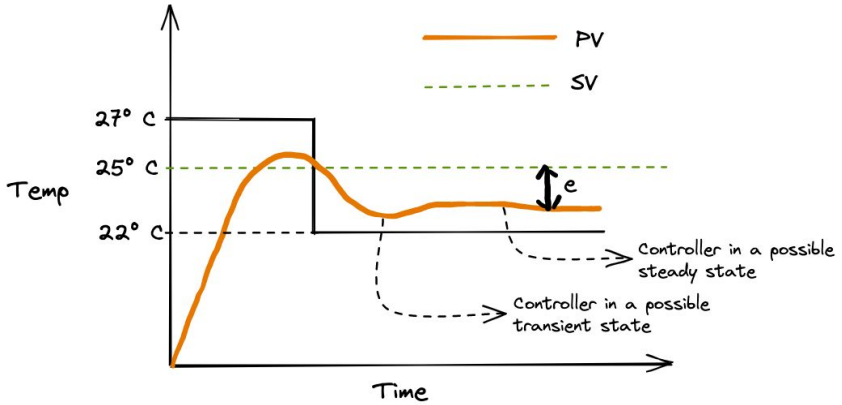
Transient and Steady State



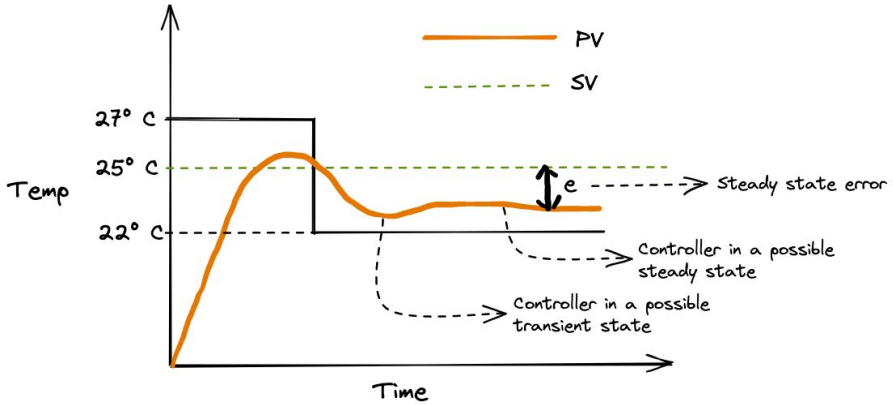
Transient and Steady State



Transient and Steady State

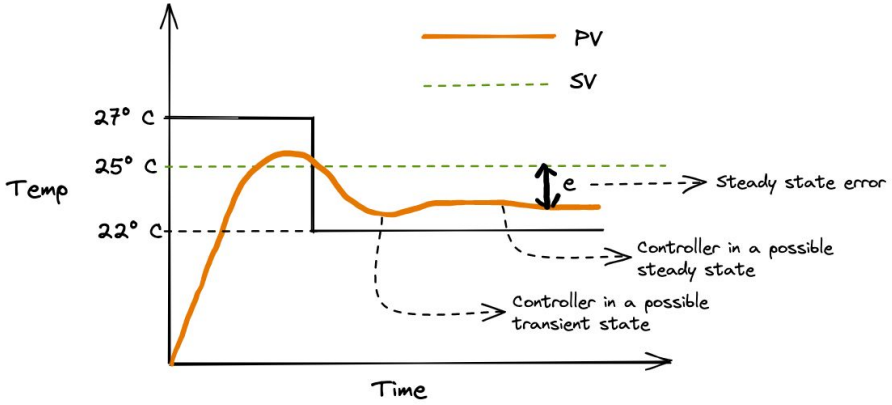


Transient and Steady State



Transient and Steady State

The lifetime of the controller can be looked at as a series of steady states “stringed” together by a series of transient states.



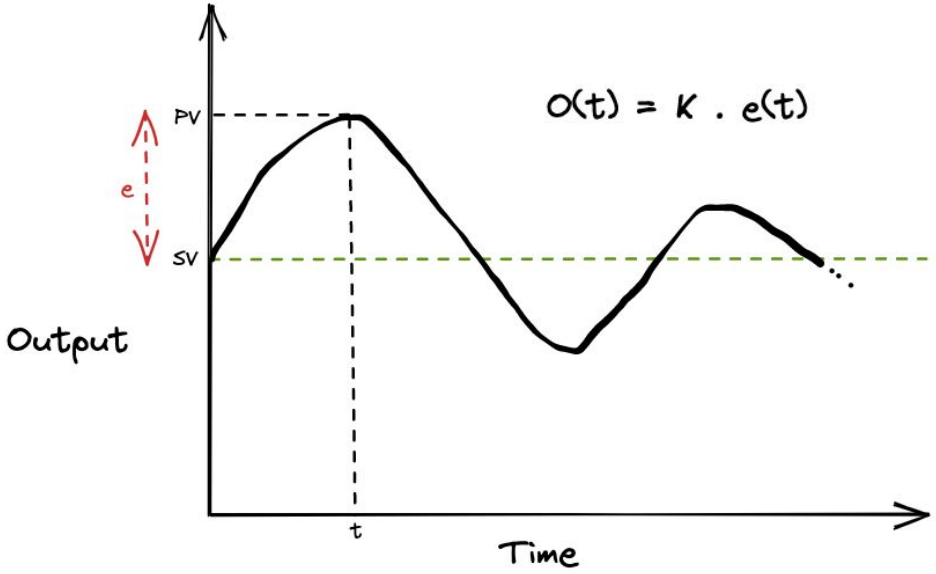
However, it’s often not this ideal. With the controller applying adjustments, the following questions come to mind:

- What if the adjustment applied overshoots or undershoots the SV?
- Can we take past experiences into account and adjust accordingly or in other words, can we compensate?
- Can we look at our current state and predict what the state is going to be in the future?

Past, Present and Future - PID Controller

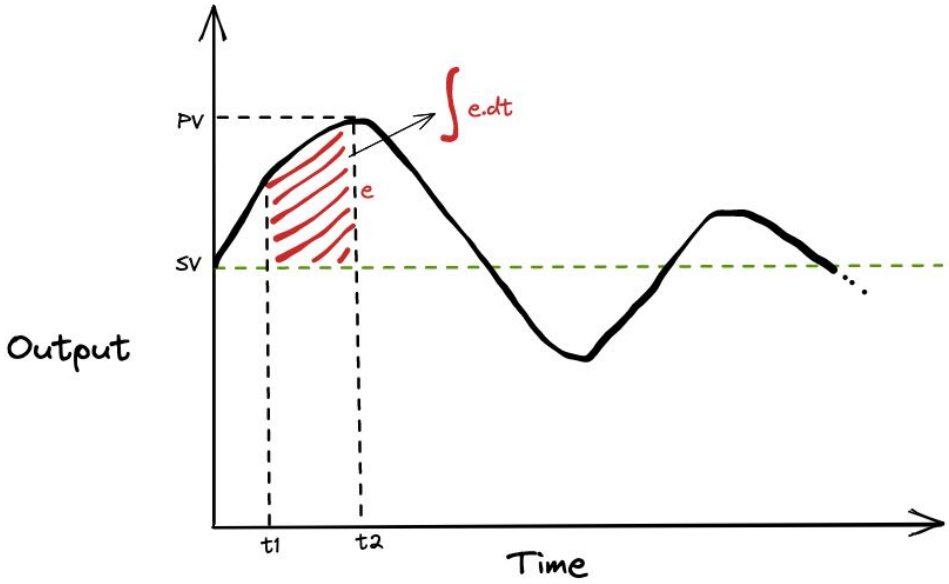
P Controller

- P - Proportional: Adjust proportional to the error
- Advantages:
  - Easy to reason about
  - Minimal state to maintain
- Disadvantages:
  - Very prone to under and over-shoot.
  - Steady state error does not converge to 0.
  - Proportional Droop



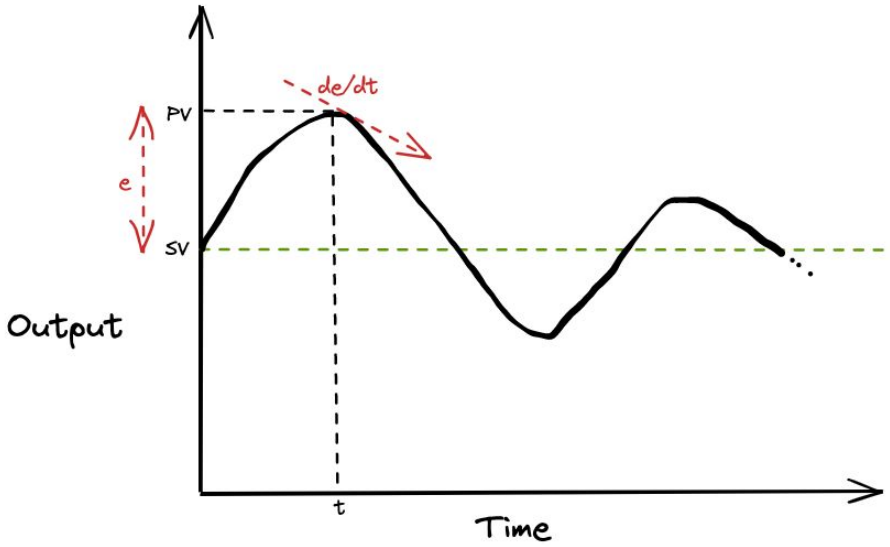
I Controller

- I - Integral: Adjust based on what the error has been in the past.
- Advantages:
  - Drives steady state error to 0.
- Disadvantages:
  - Prone to under and over-shoot.



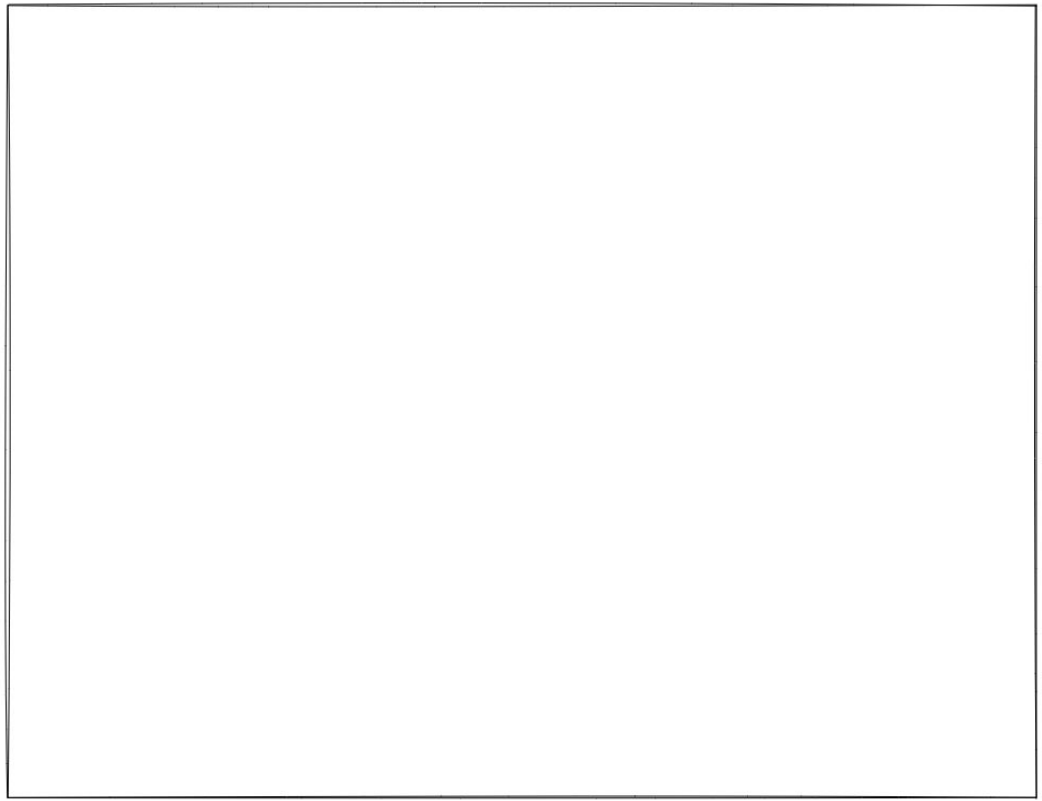
D Controller

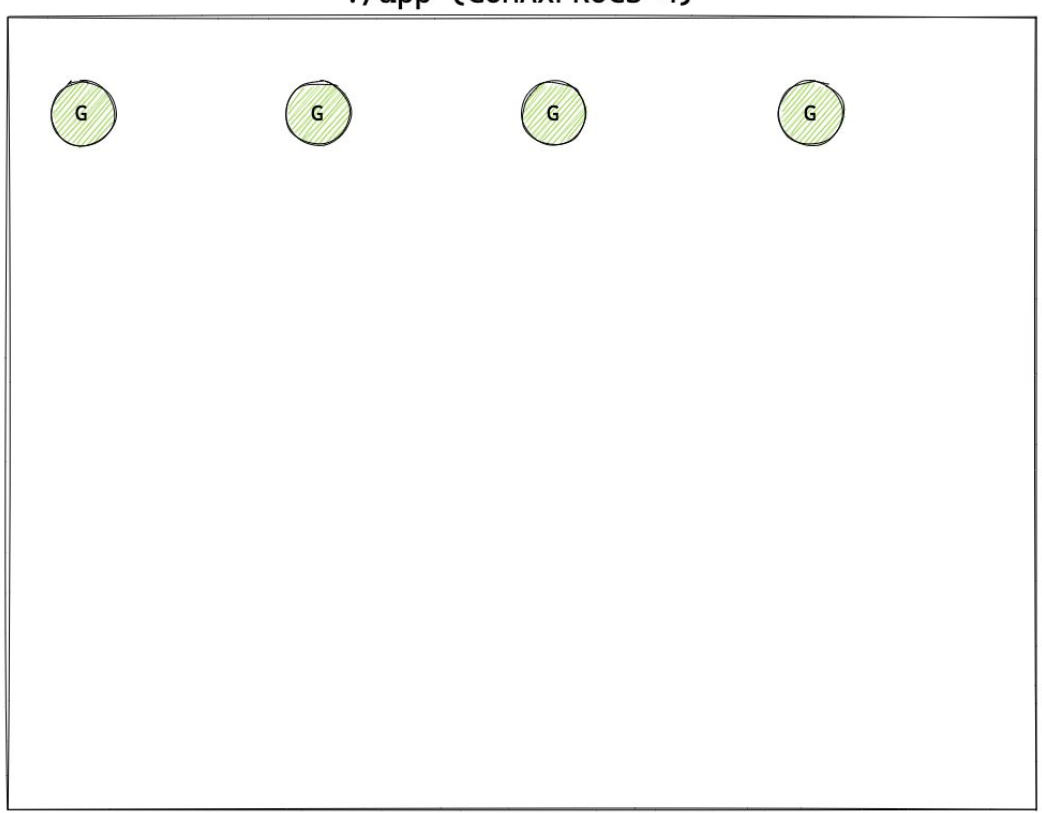
- D - Derivative: Adjust based on how the error is changing (anticipate the future).
- Advantages:
  - Great for applying corrective actions.
  - Speeds up time taken to reach steady state.
- Disadvantages:
  - Highly sensitive to noise.

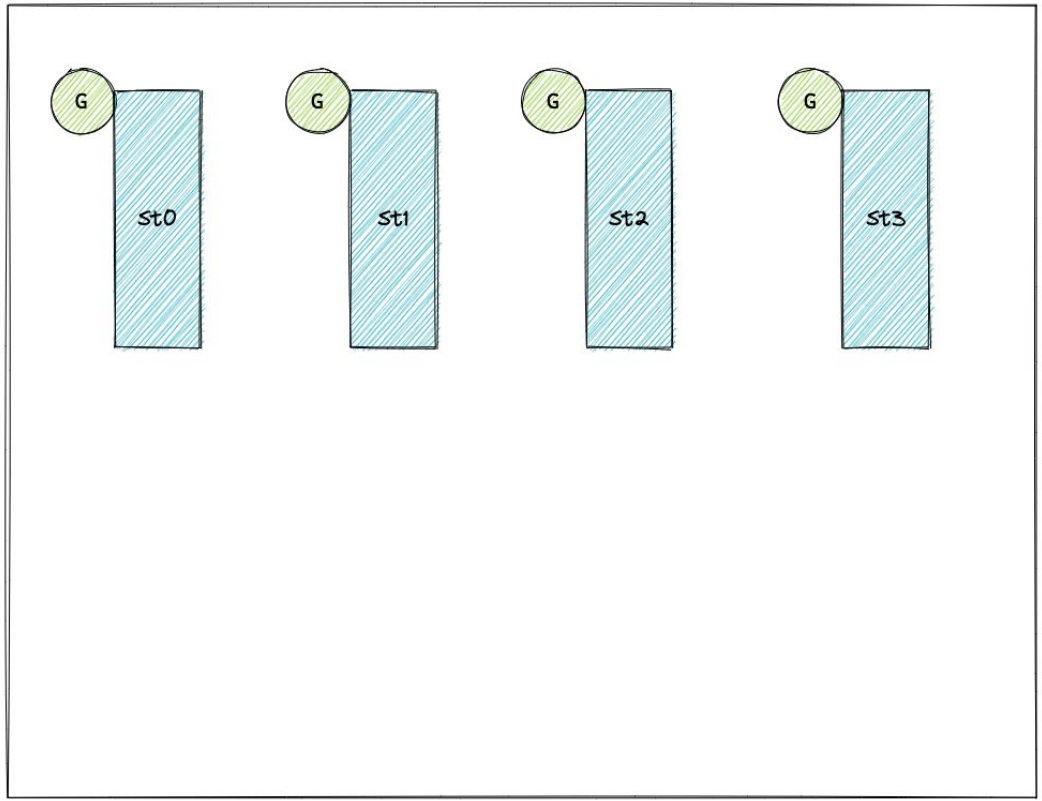


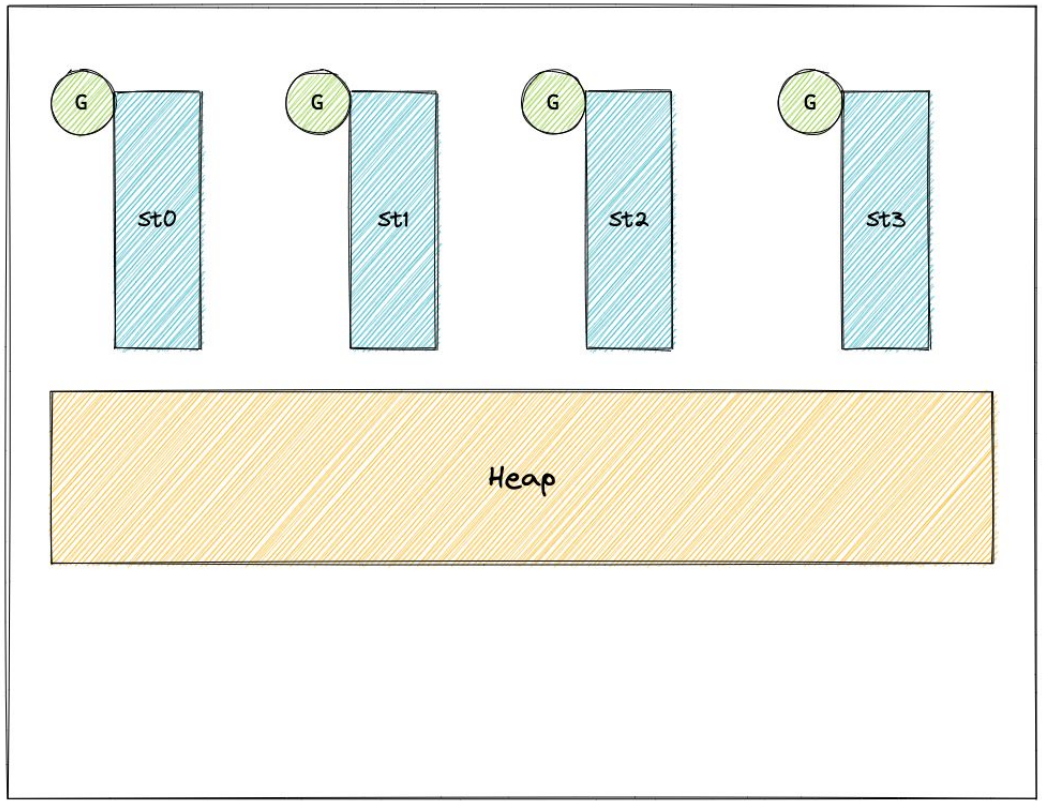
What does a typical Go application look like?

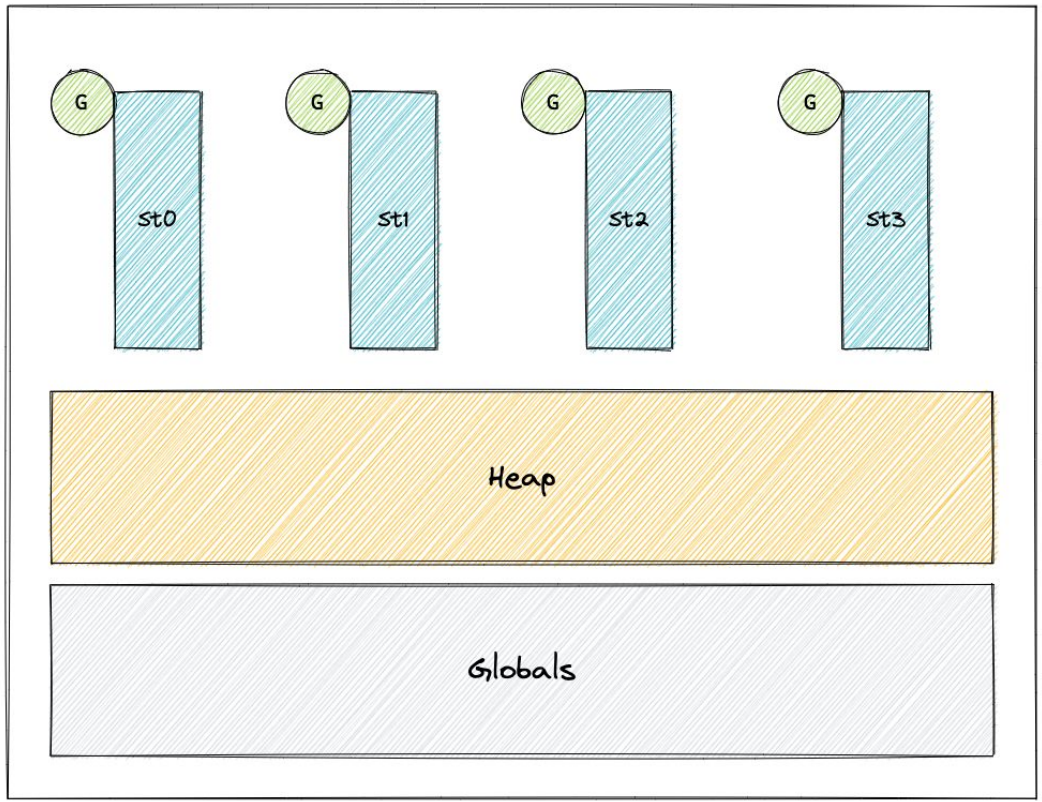
./app (GOMAXPROCS=4)

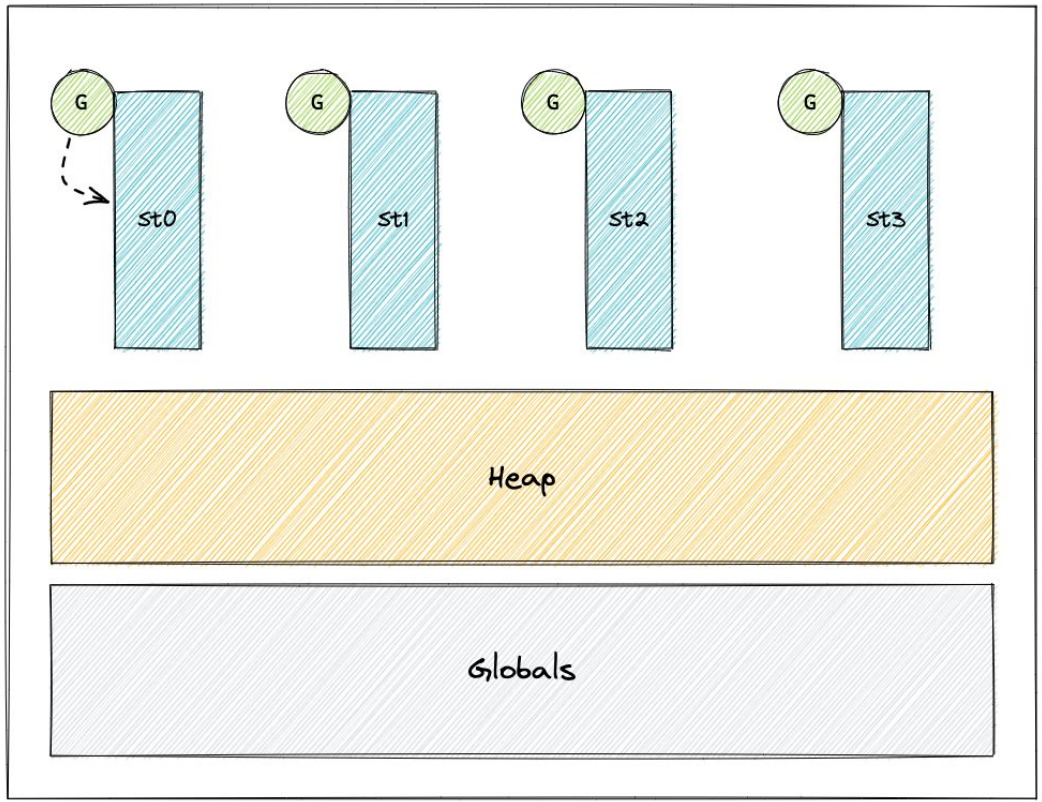


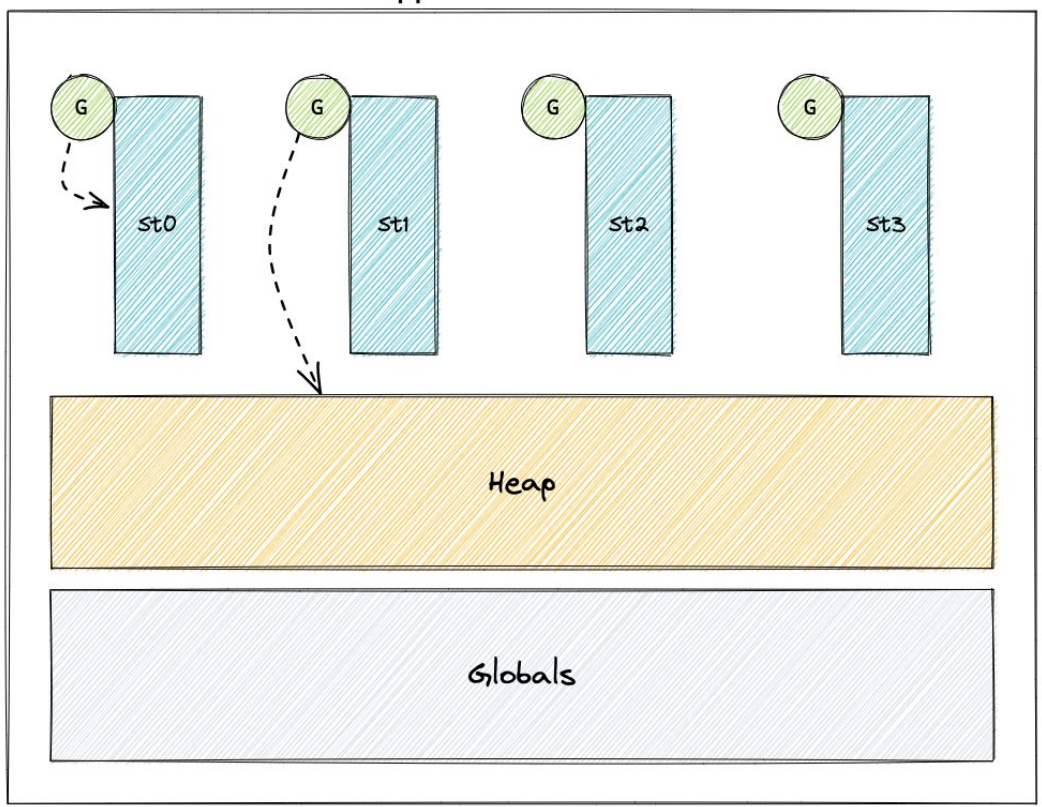

./app (GOMAXPROCS=4)
G
G
G
G

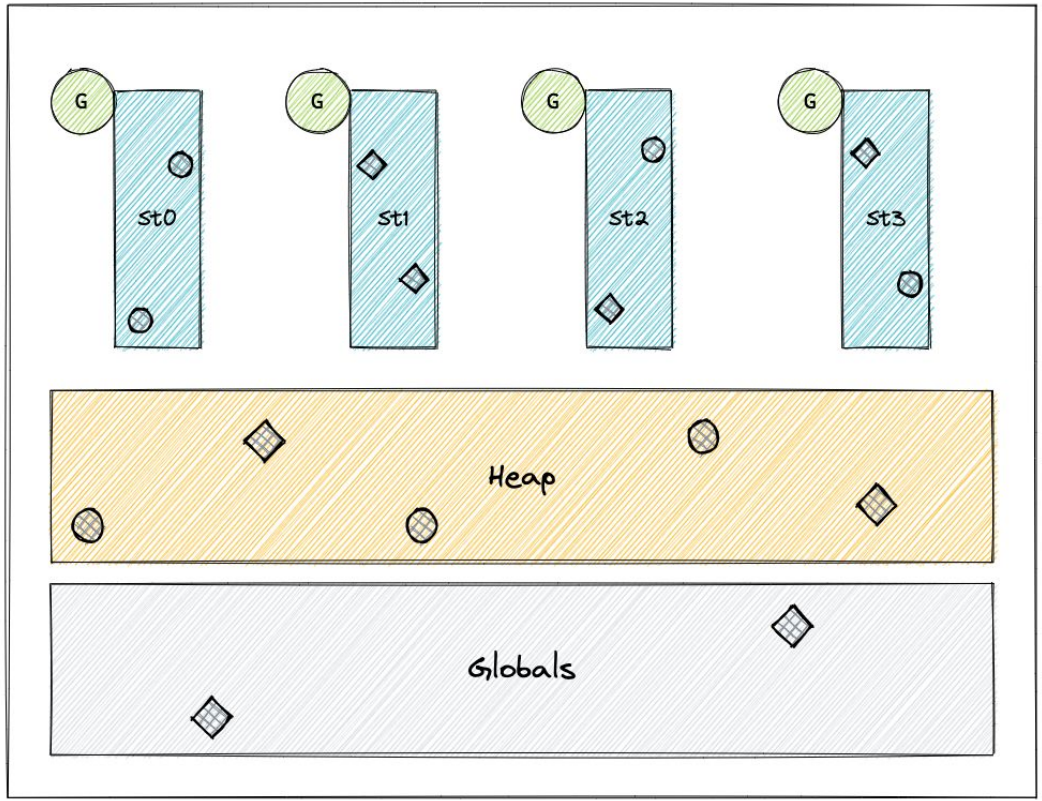

./app (GOMAXPROCS=4)
G
St0
G
St1
G
St2
G
St3


./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap


./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap
Globals

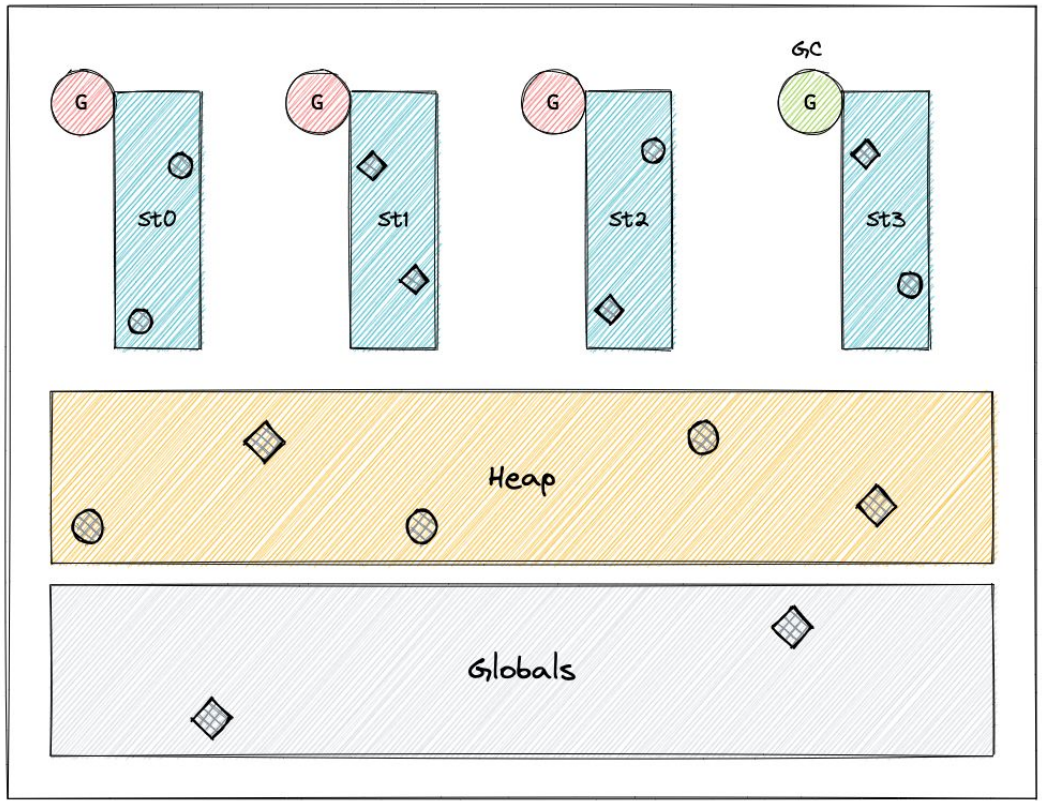

./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap
Globals

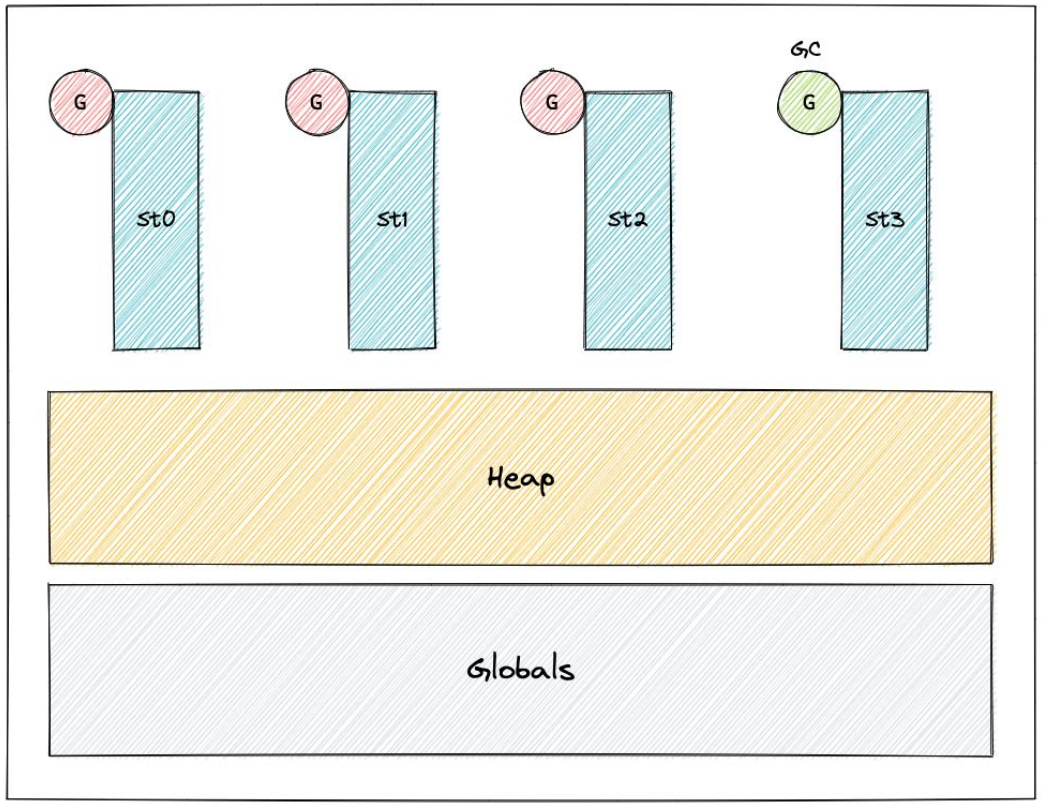

./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap
Globals

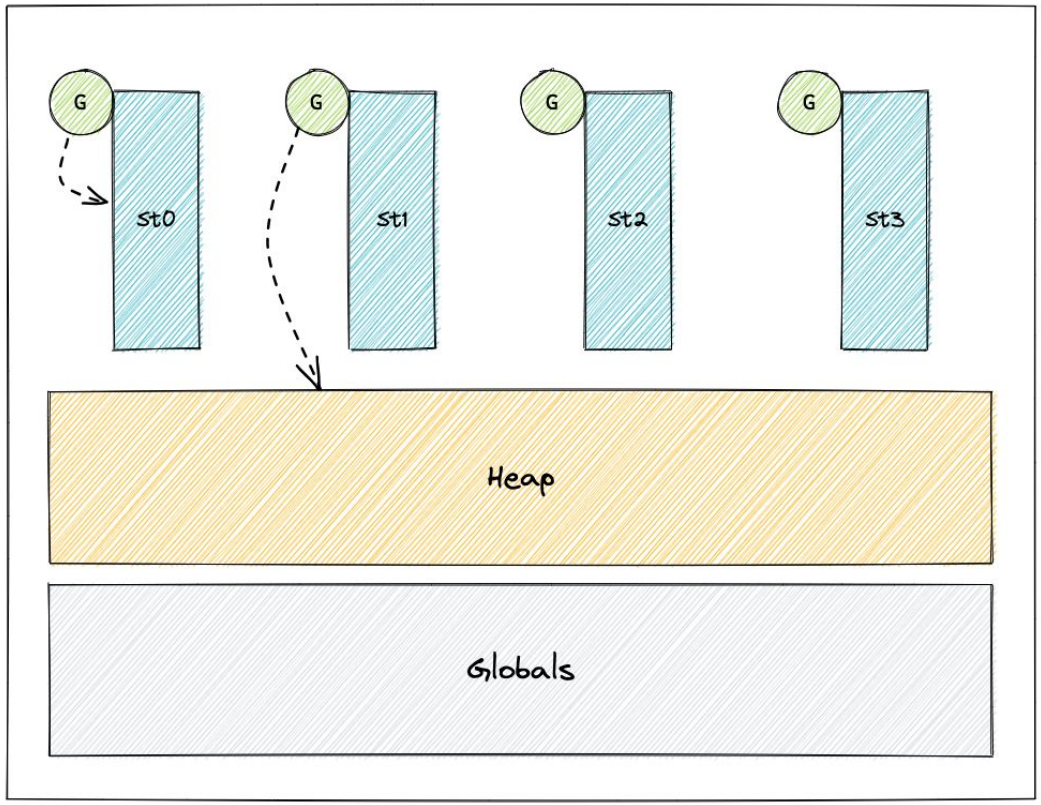

./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap
Globals

It’s time to cleanup! How can we go about doing that?

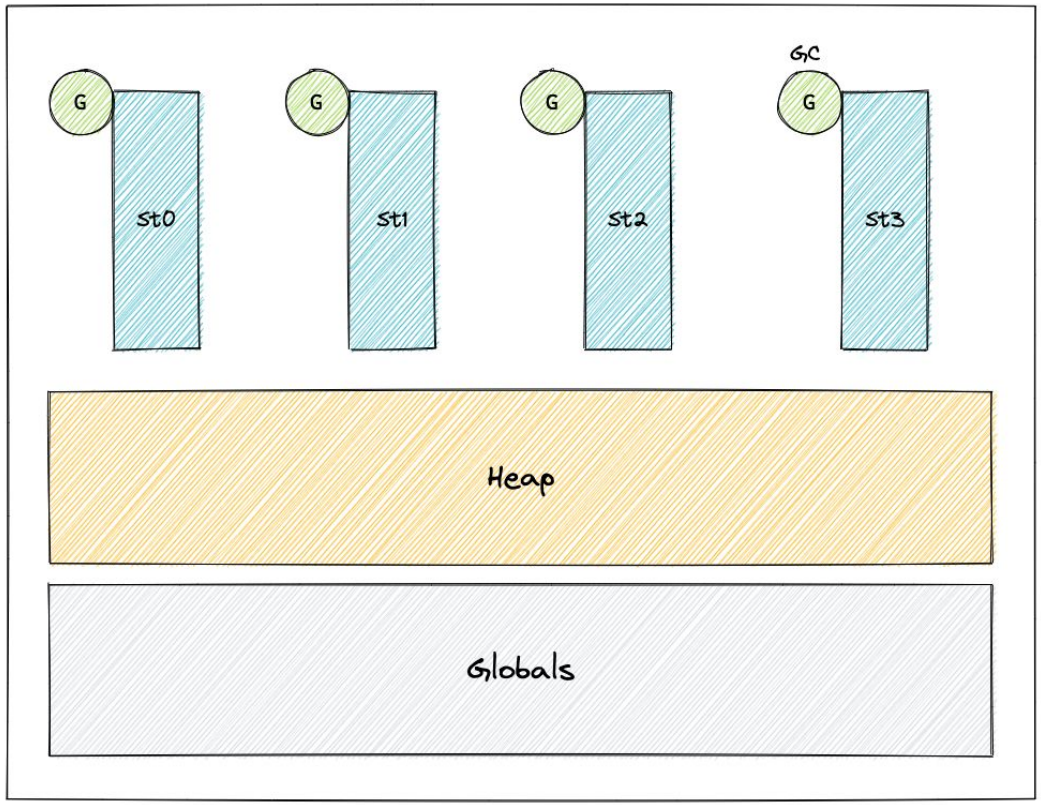
Hmm… One way could be to stop the application and do a sweep.


./app (GOMAXPROCS=4)
G
G
G
GC
G
St0
St1
St2
St3
Heap
Globals

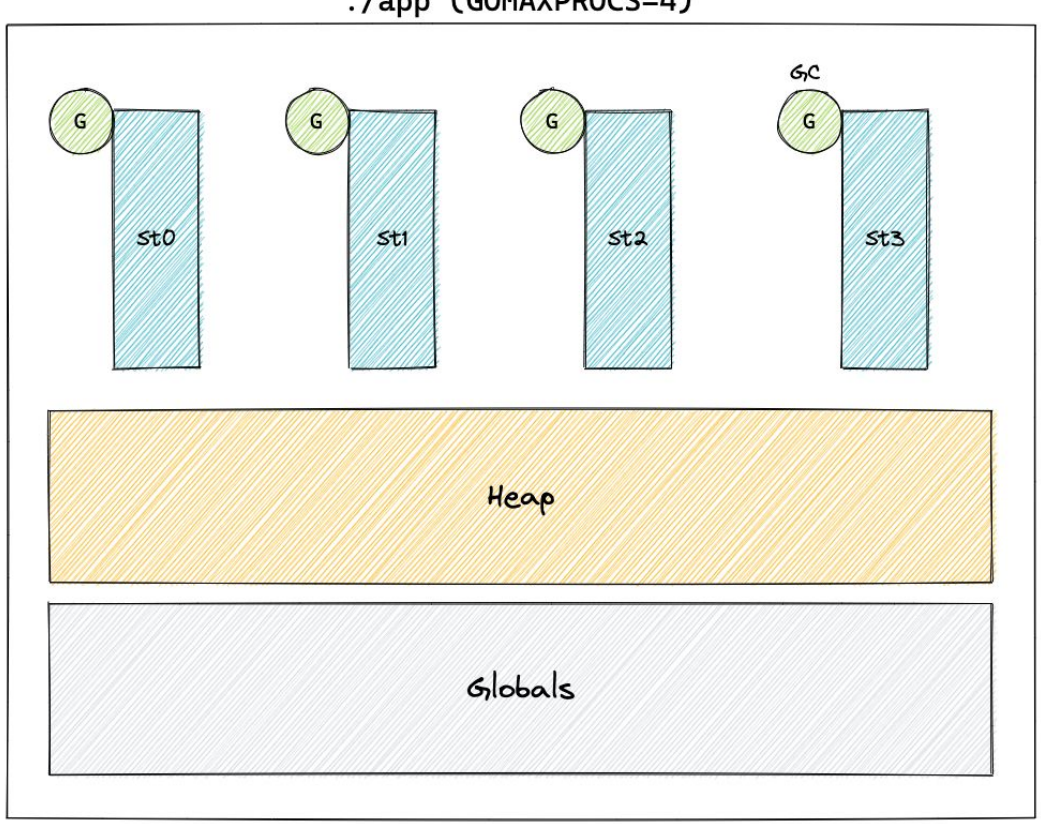

./app (GOMAXPROCS=4)
G
G
G
GC
G
St0
St1
St2
St3
Heap
Globals


./app (GOMAXPROCS=4)
G
G
G
G
St0
St1
St2
St3
Heap
Globals

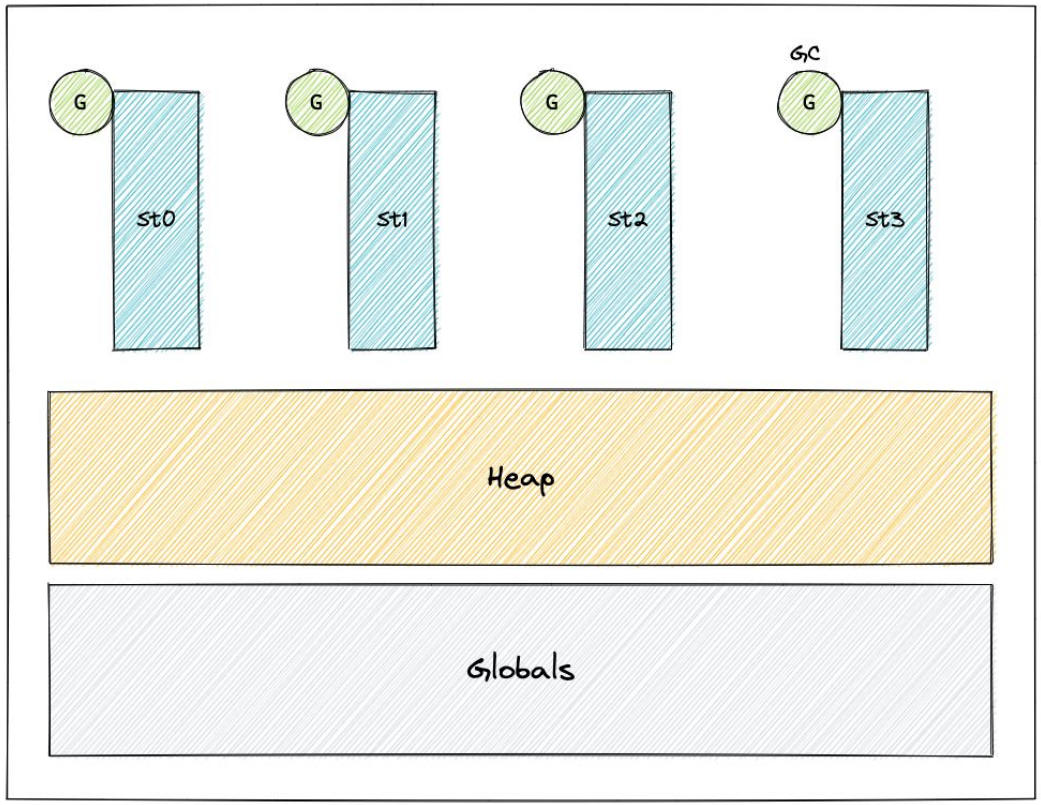
Another way could be to do things concurrently…


./app (GOMAXPROCS=4)
G
St0
G
St1
G
St2
GC
G
St3
Heap
Globals

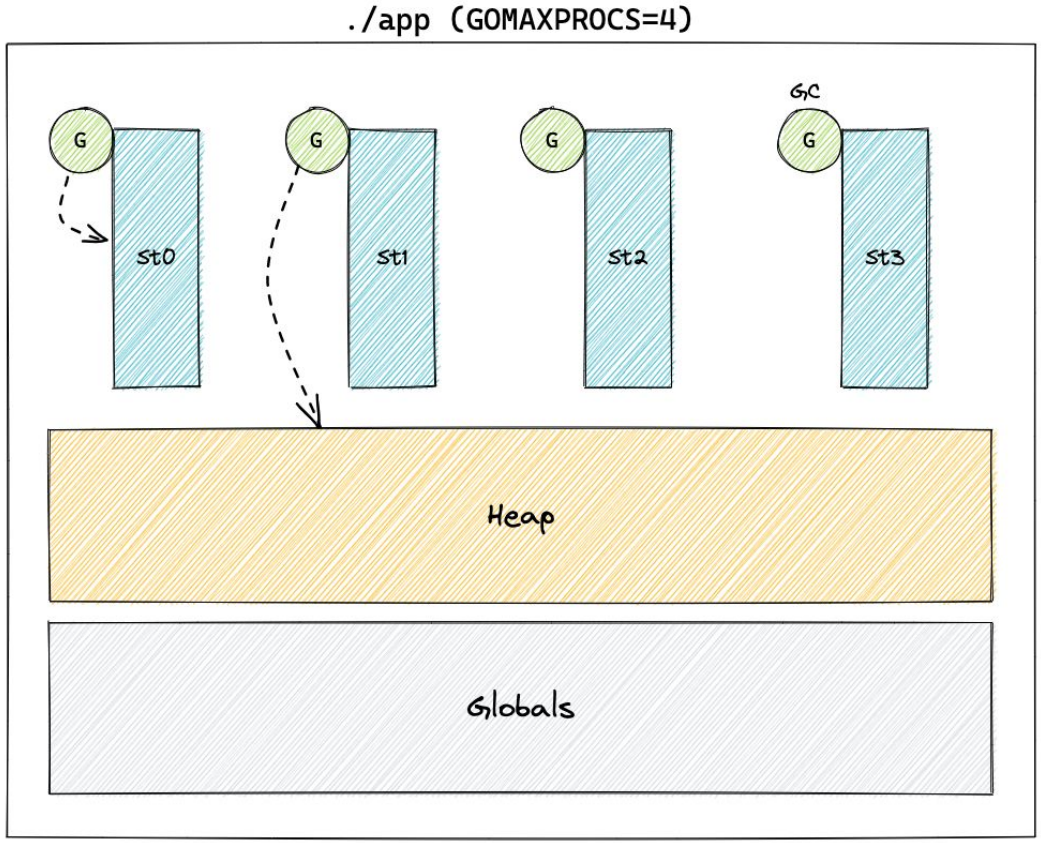
Go does GC this way!



But...

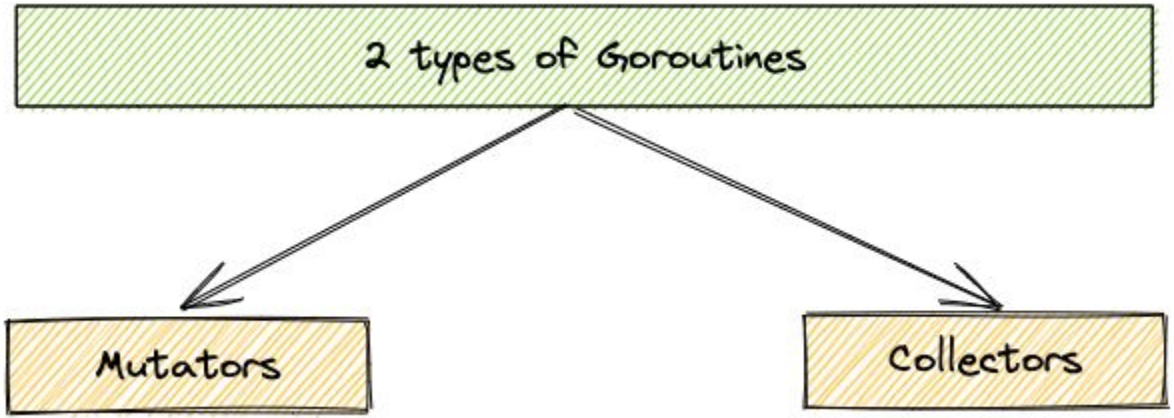


Application is still allocating!

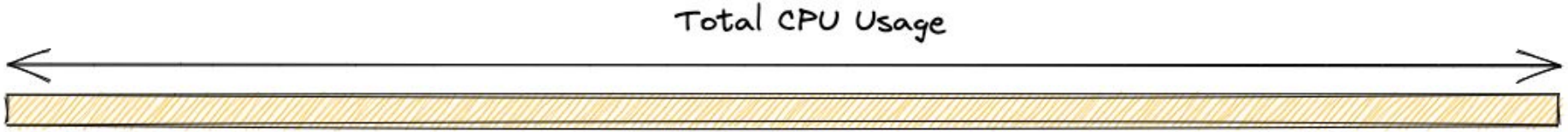


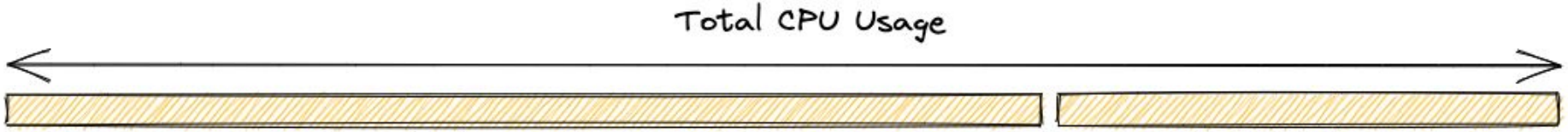
So, how does Go do GC?

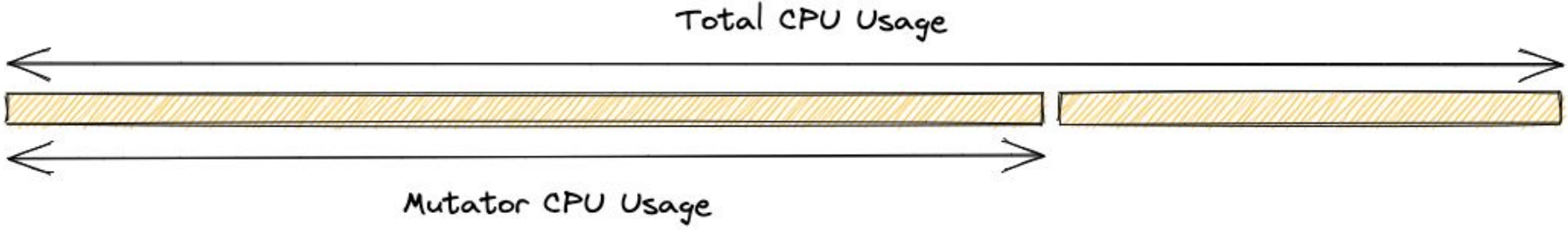
Interlude

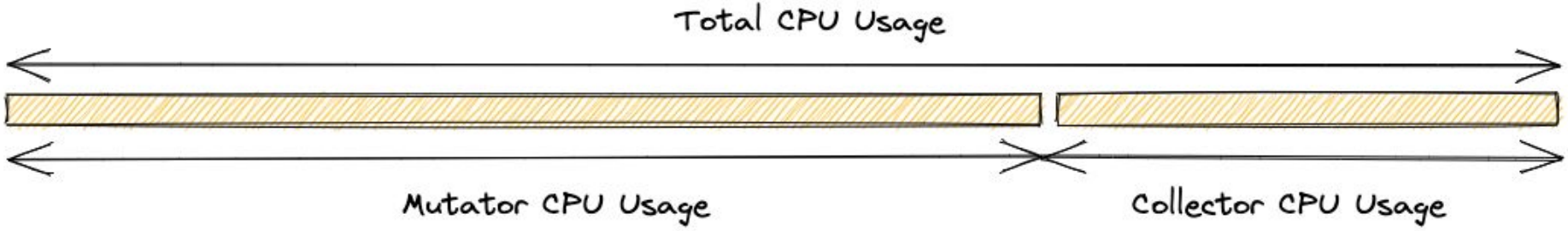


Considering mutators and collectors will run together at some point, there is a fundamental tradeoff involved, let’s consider the following:


Total CPU Usage


Total CPU Usage

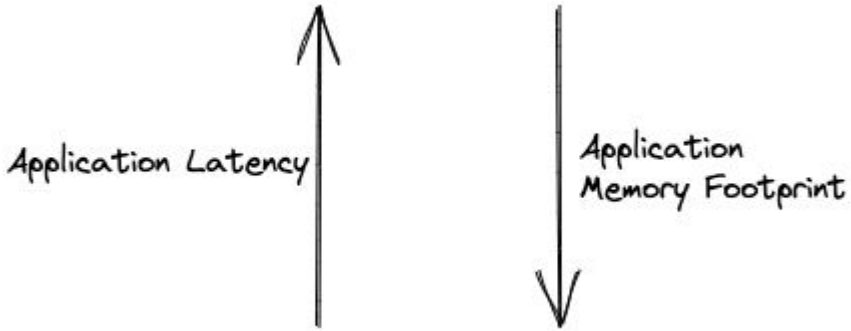

Total CPU Usage
Mutator CPU Usage


Total CPU Usage
Mutator CPU Usage
Collector CPU Usage

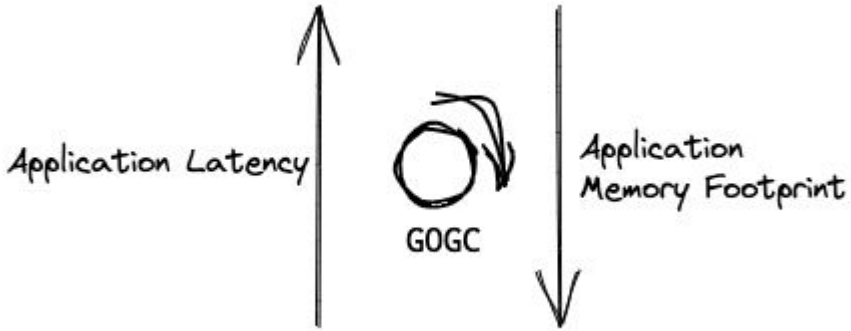
Over a period of time (and normal circumstances), this tradeoff translates to:

- More collector CPU Usage => More time spent doing GC work => Potentially lower mem. footprint for that time period.
  - But: less time given to application => Higher application latencies.
- More mutator CPU Usage => More time given to application => Potentially lower application latencies for that time period
  - But: less time spent doing GC work => Higher mem. footprint.

Over a period of time (and normal circumstances), this tradeoff translates to:



Over a period of time (and normal circumstances), this tradeoff translates to:



GOGC

GOGC

- Let $H_m(n)$ be the size of the objects that were marked live after cycle n.
  - “Objects” is intentionally vague for now!

GOGC

- Let $H_m(n)$ be the size of the objects that were marked live after cycle n.
  - “Objects” is intentionally vague for now!
- Let $H_g(n)$ be the value to which we are willing to let the memory footprint grow before we start a GC cycle.

GOGC

- Let $H_m(n)$ be the size of the objects that were marked live after cycle n.
  - “Objects” is intentionally vague for now!
- Let $H_g(n)$ be the value to which we are willing to let the memory footprint grow before we start a GC cycle.
  - Or in other words, this is the *heap goal*.

GOGC

- Let $H_m(n)$ be the size of the objects that were marked live after cycle n.
  - “Objects” is intentionally vague for now!
- Let $H_g(n)$ be the value to which we are willing to let the memory footprint grow before we start a GC cycle.
  - Or in other words, this is the *heap goal*.

$$H_g(n) = H_m(n-1) \times [1 + GOGC/100]$$

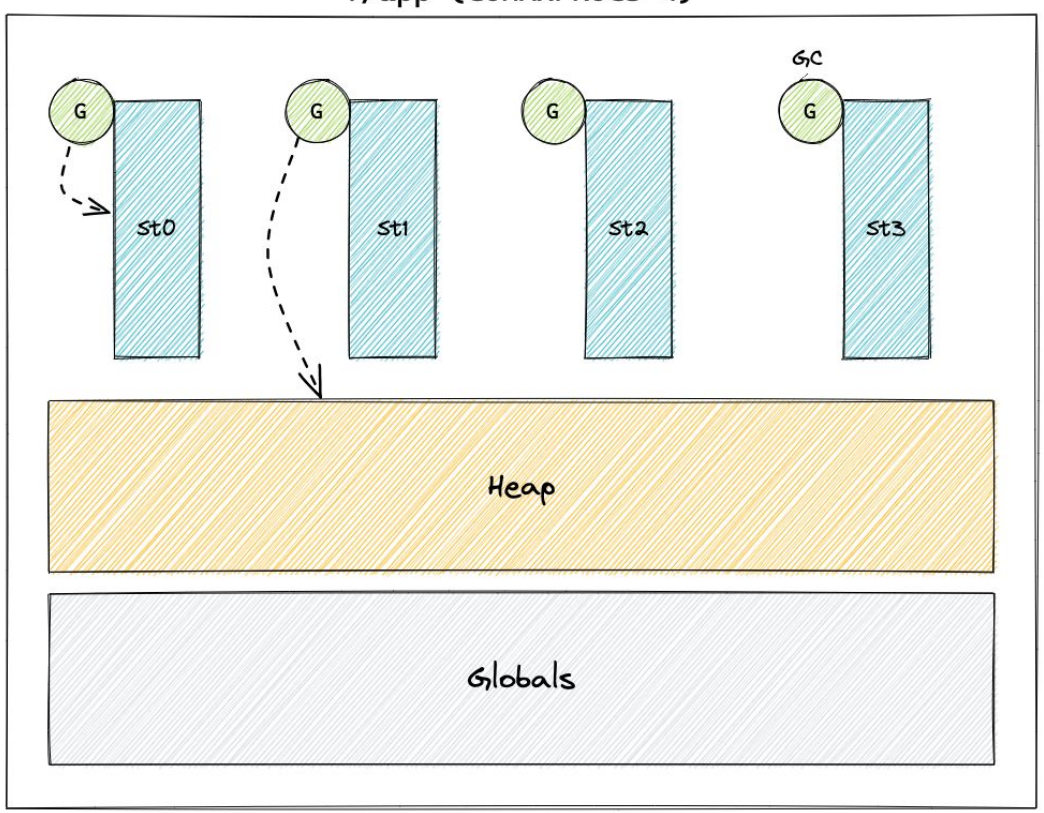
GOGC

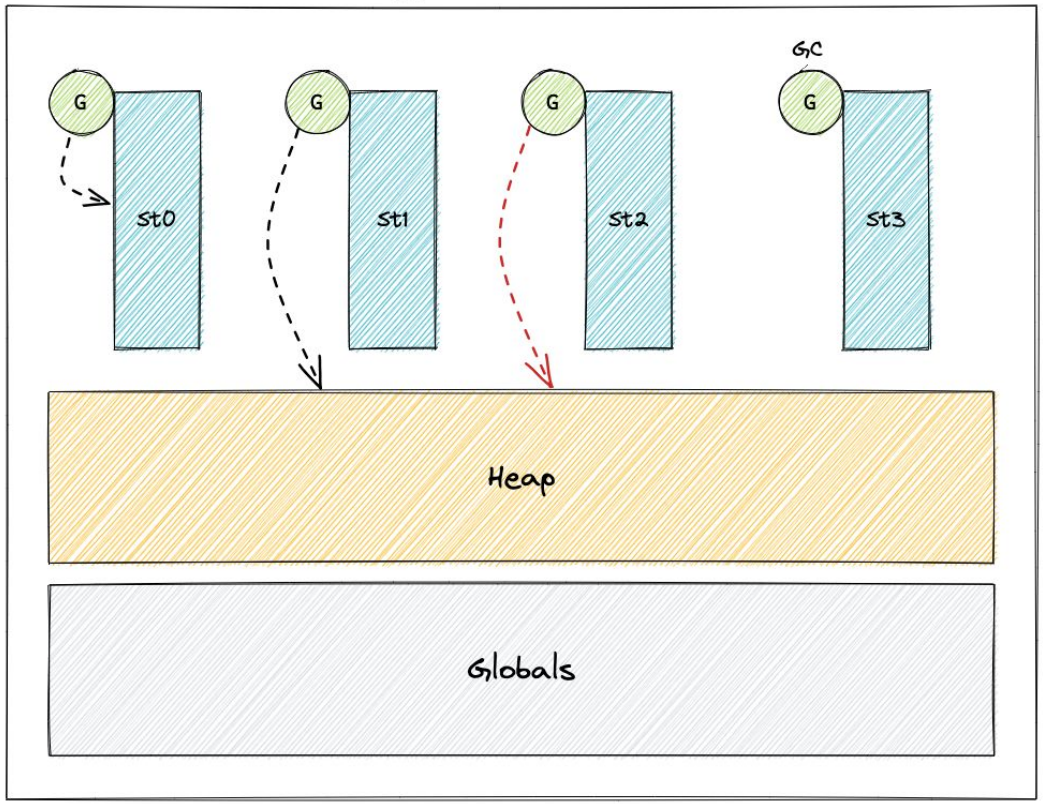
“The GOGC variable sets the initial garbage collection target percentage. A collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage.”

https://pkg.go.dev/runtime

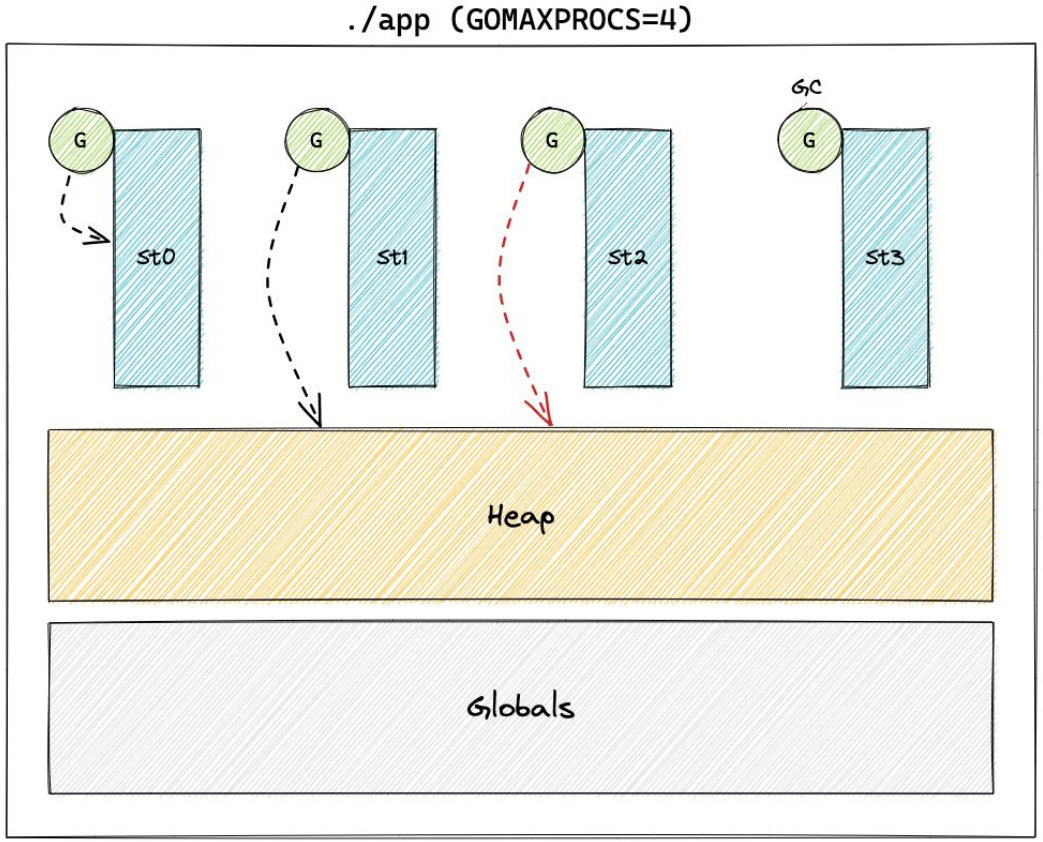
Interlude: GC Mark Assists

Let’s take a look at our app again

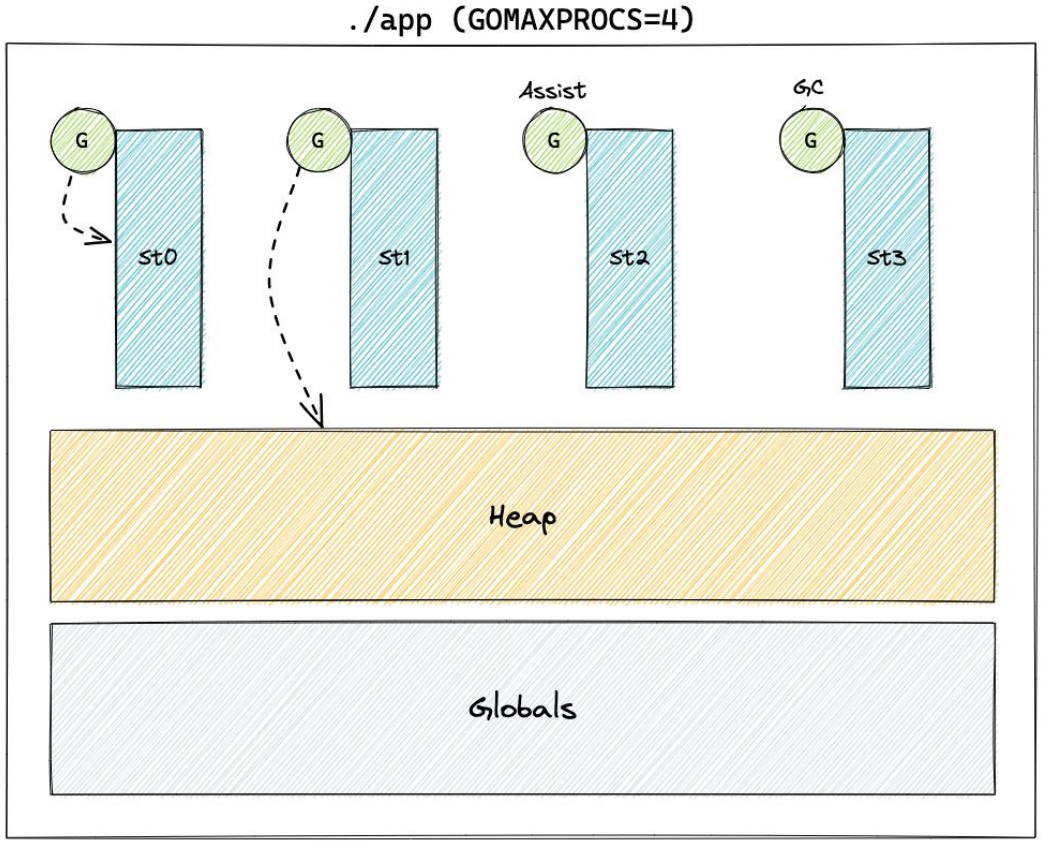

./app (GOMAXPROCS=4)
G
G
G
GC
G
St0
St1
St2
St3
Heap
Globals


./app (GOMAXPROCS=4)
G
G
G
GC
G
St0
St1
St2
St3
Heap
Globals

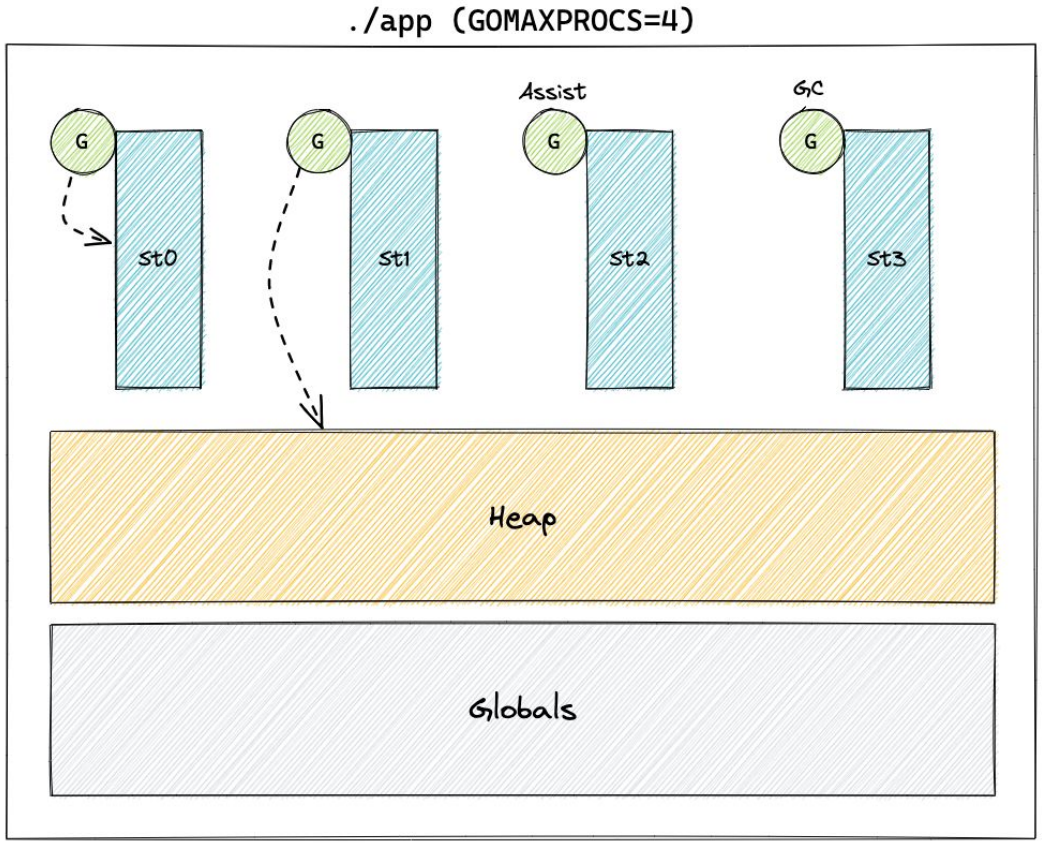
- If a goroutine starts allocating faster than the GC can keep up, it can lead to unbounded heap growth.



- If a goroutine starts allocating faster than the GC can keep up, it can lead to unbounded heap growth.
- To deal with this, the GC asks this goroutine to *assist* with *marking*.



Mark assists are a *reactive* measure to keep memory usage stable in unstable conditions, at the expense of CPU time.



The question we now need to try and answer is: *when* do we start a GC cycle?

When do we start a GC cycle?

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
    - No allocations during GC

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
  - No allocations during GC
  - We could start a cycle when the $H_m(n-1) = H_g(n)$

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
  - No allocations during GC
  - We could start a cycle when the $H_m(n-1) = H_g(n)$
- But now, mutators and collectors run concurrently
  - Allocations happen during the concurrent marking phase

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
  - No allocations during GC
  - We could start a cycle when the $H_m(n-1) = H_g(n)$
- But now, mutators and collectors run concurrently
  - Allocations happen during the concurrent marking phase
  - How do we still respect $H_g(n)$?

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
  - No allocations during GC
  - We could start a cycle when the $H_m(n-1) = H_g(n)$
- But now, mutators and collectors run concurrently
  - Allocations happen during the concurrent marking phase
  - How do we still respect $H_g(n)$?
  - We need to start early!
  - How early?

When do we start a GC cycle?

- Prior to Go 1.5, the GC was a parallel STW collector.
  - No allocations during GC
  - We could start a cycle when the $H_m(n-1) = H_g(n)$
- But now, mutators and collectors run concurrently
  - Allocations happen during the concurrent marking phase
  - How do we still respect $H_g(n)$?
  - We need to start early!
  - How early?

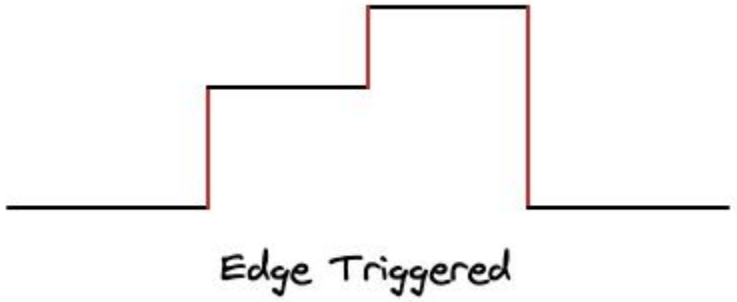
That's a question for the ***GC Pacer.***

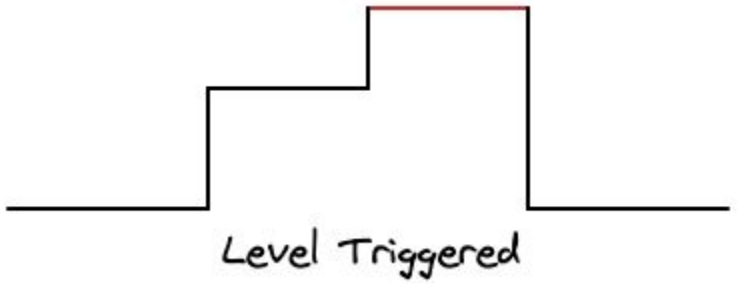
The GC Pacer Has 2 Fundamental Goals

1. Maintain a target GC CPU utilization.
2. Get the size of the live heap as close to the heap goal as possible.

Note:

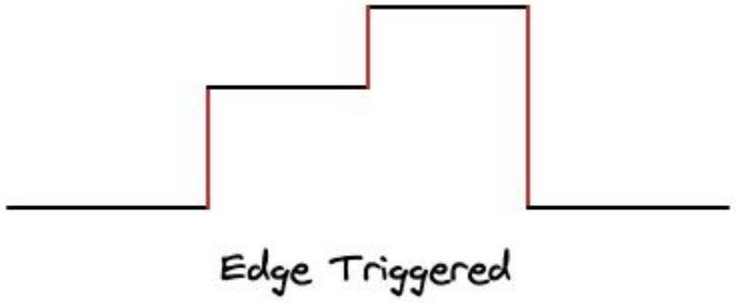
- It helps thinking of the Pacer as a level-triggered system as opposed to an edge-triggered one.

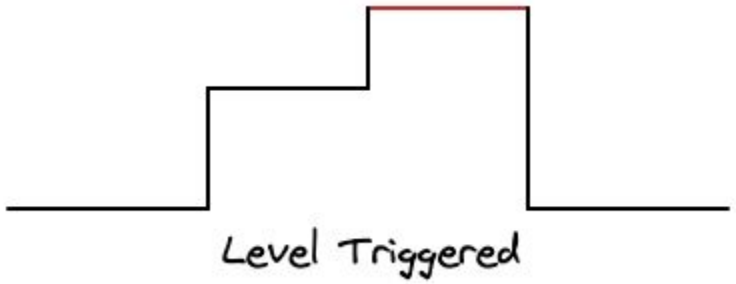




Note:

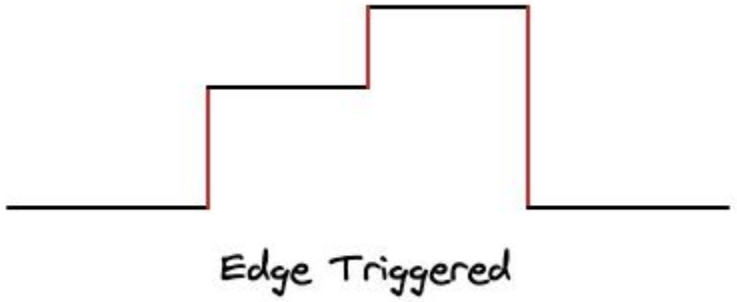
- It helps thinking of the Pacer as a level-triggered system as opposed to an edge-triggered one.
    a. The Pacer concerns itself with a macro view of the system, and cares about how the behaviour is aggregating over a period of time.

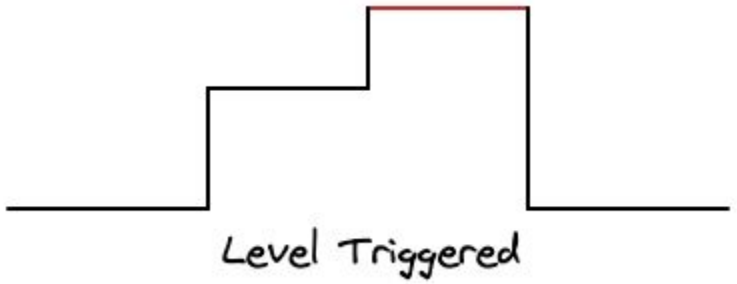




Note:

- It helps thinking of the Pacer as a level-triggered system as opposed to an edge-triggered one.
    a. The Pacer concerns itself with a macro view of the system, and cares about how the behaviour is aggregating over a period of time.
    b. It does *not* concern itself with moment-to-moment, individual allocations.





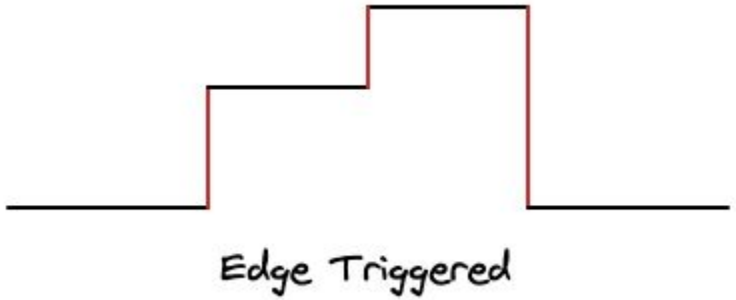
Note:

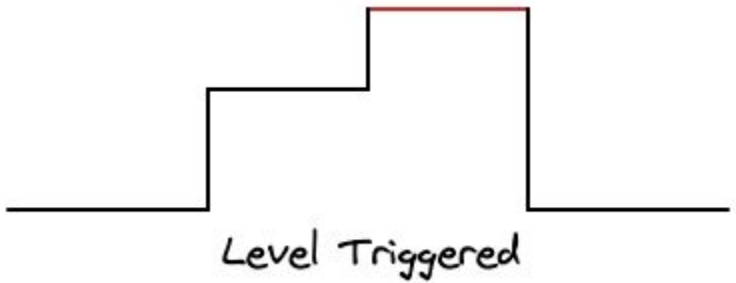
- It helps thinking of the Pacer as a level-triggered system as opposed to an edge-triggered one.

Instead, it *expects* the application to settle on some state:

✨The steady state ✨

The goals of the pacer are defined for this steady state.





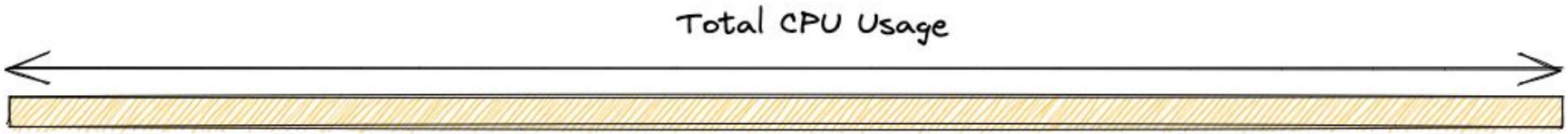
✨ Steady State ✨

- Constant allocation rate
- Constant heap size
- Constant heap composition

GC Pacer Prior To Go 1.18

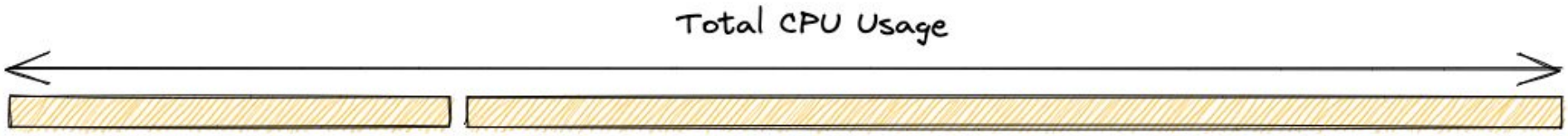
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



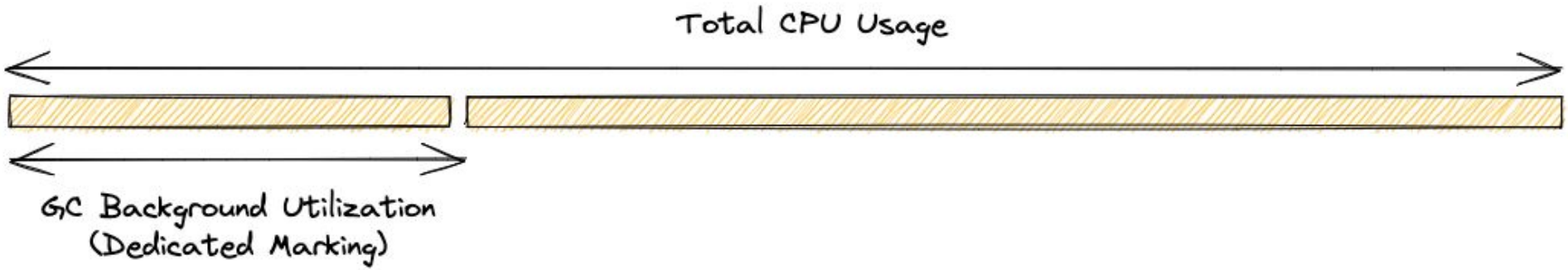
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



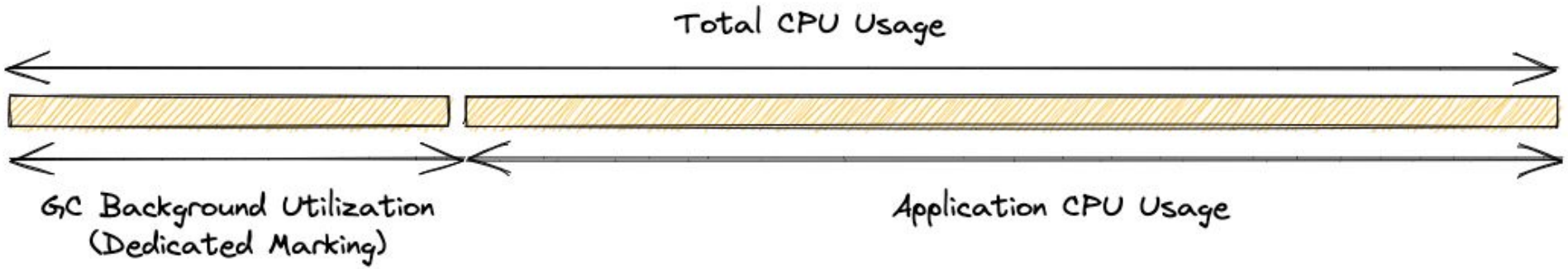
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



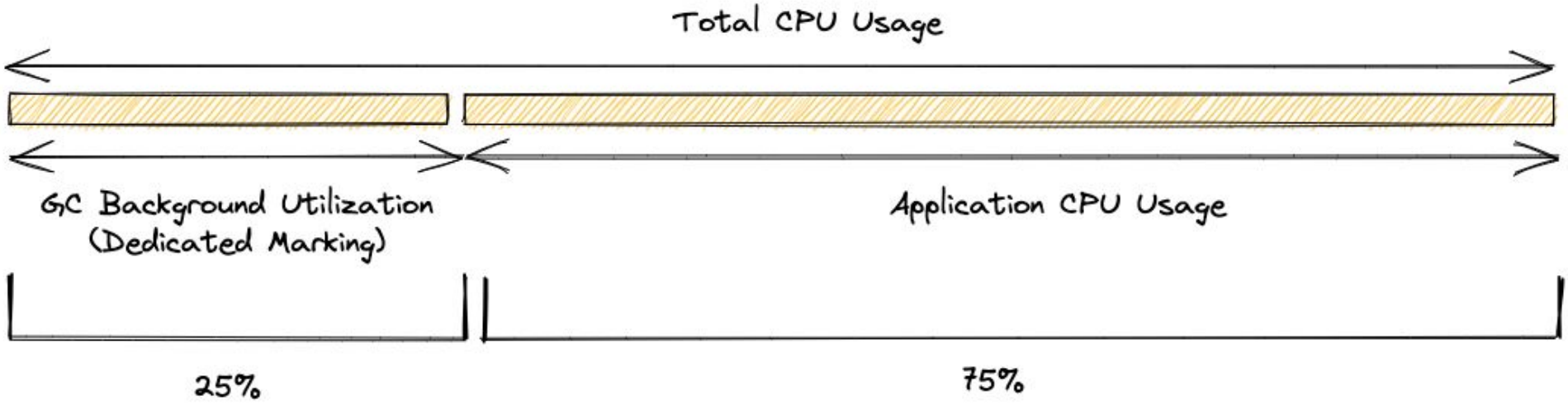
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



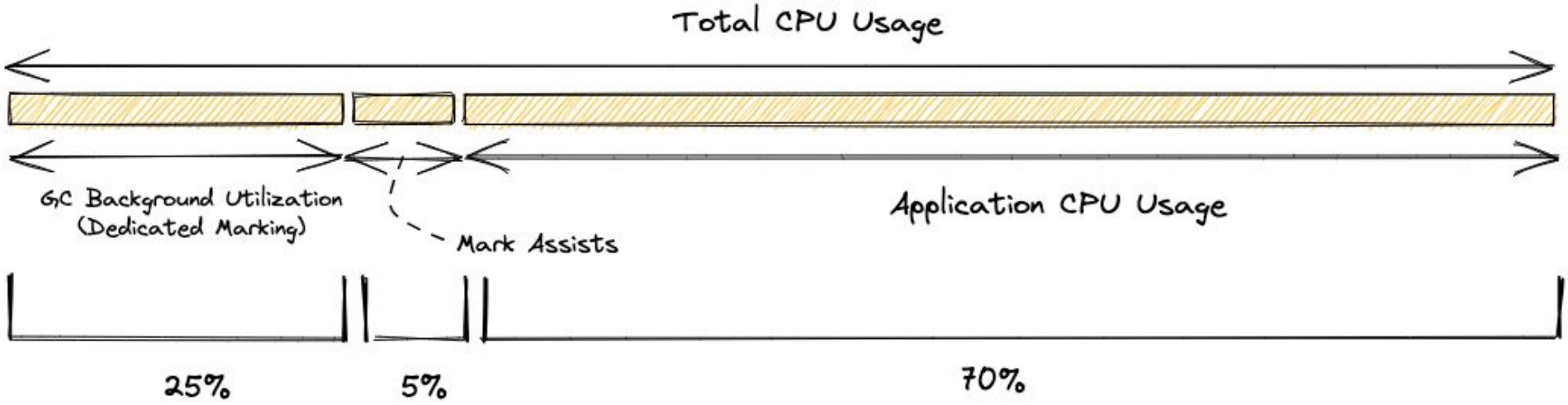
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



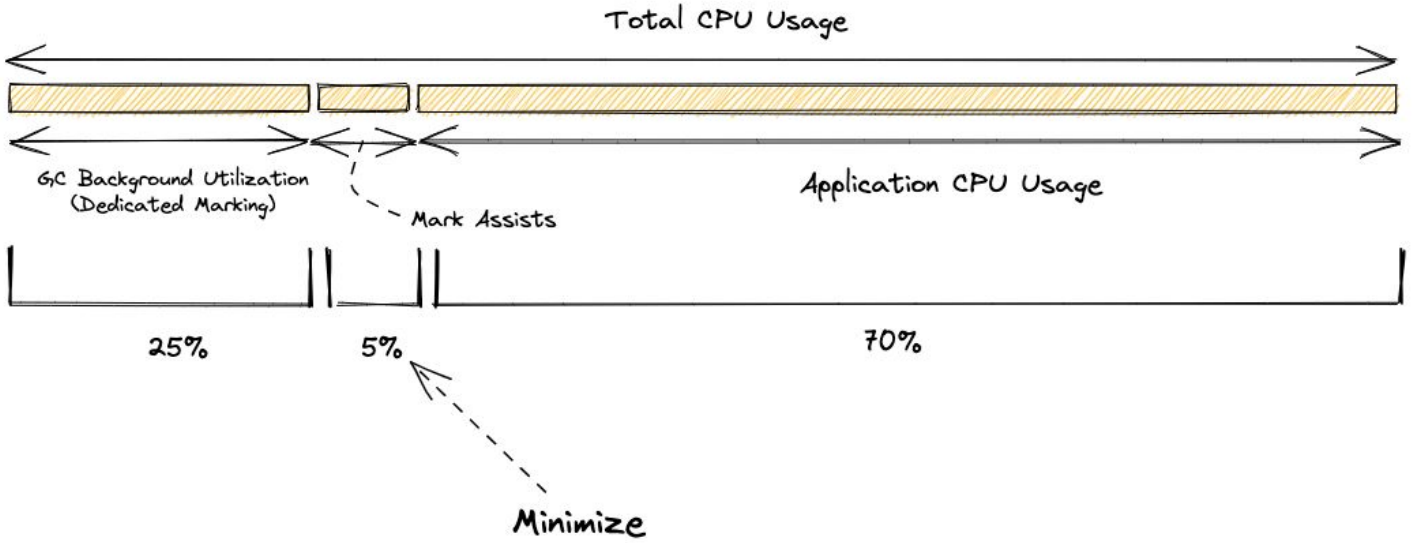
GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization



GC Pacer Prior To Go 1.18

Optimization Goal 1: Target CPU Utilization

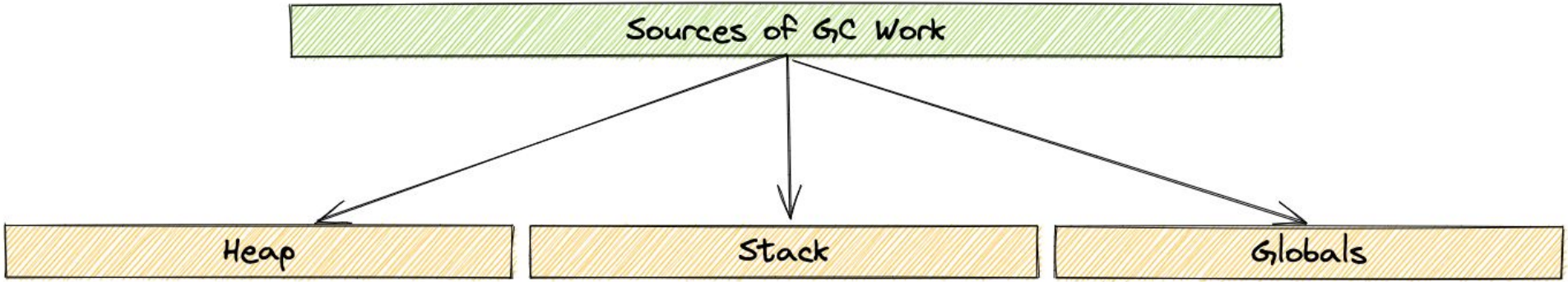


GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.

GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.



GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.

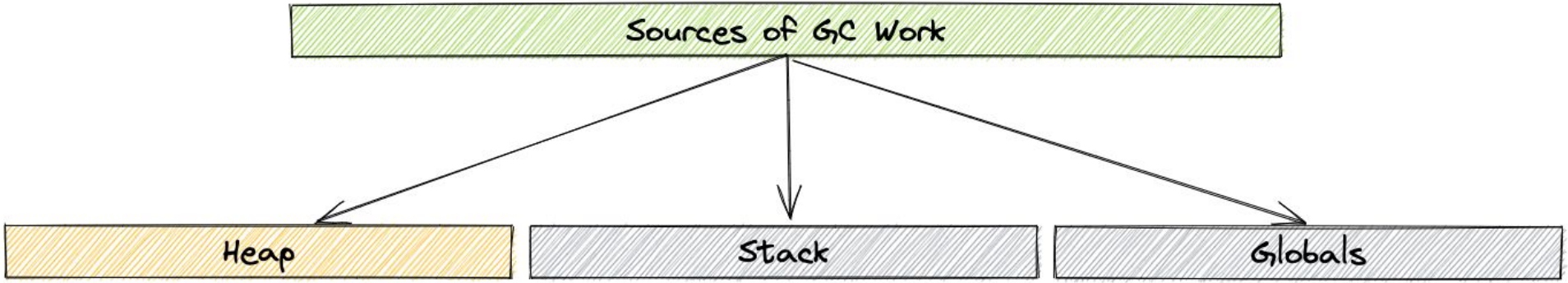
- From our earlier GOGC discussion:

$$H_g(n) = H_m(n-1) \times [1 + GOGC/100]$$

- $H_m(n-1)$ here is the amount of ***heap*** memory marked live after cycle n.
- We do not take into account other sources of GC work.
    - Assume they are negligible compared to the heap.

GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.

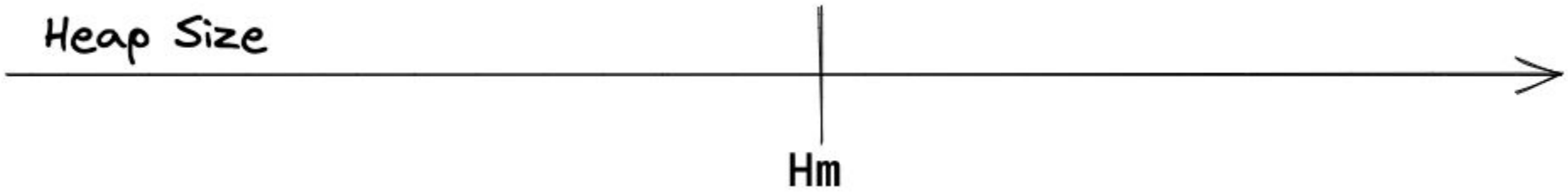


GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.

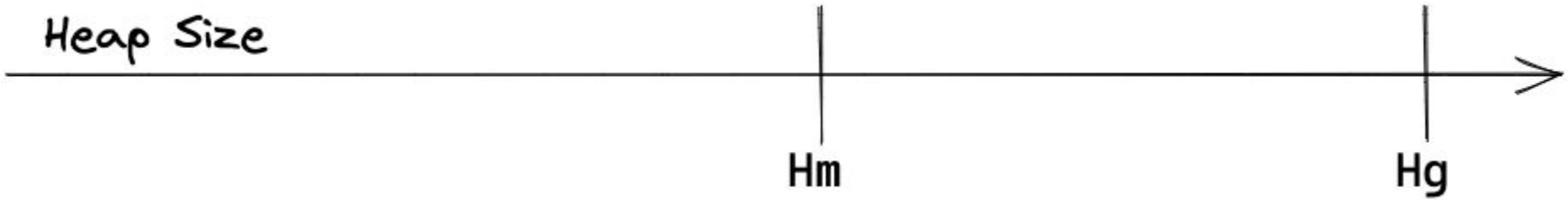
GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.



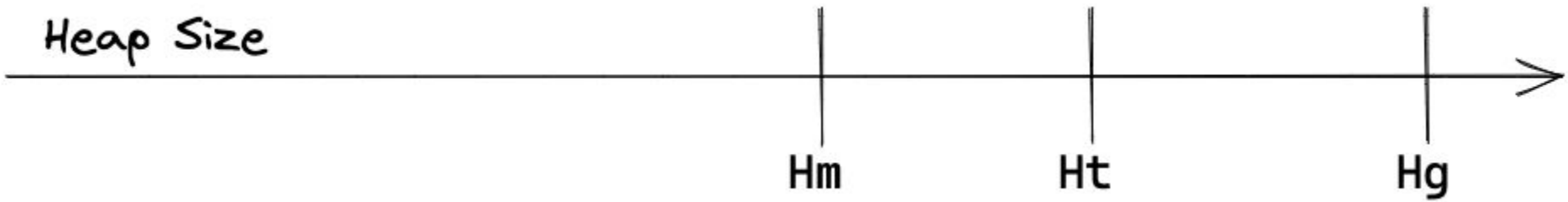
GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.



GC Pacer Prior To Go 1.18

Optimization Goal 2: Get the size of the live heap as close to the heap goal as possible.



GC Pacer Prior To Go 1.18

- H_t is the size of the heap at which we to trigger a GC cycle.
- We determine this value by using our optimization goals as “guides”
    - Or more formally - constraints.
- We know where we are currently, we know where we’d like to be - given this, how do we compute H_t?

GC Pacer Prior To Go 1.18

- H_t is the size of the heap at which we to trigger a GC cycle.
- We determine this value by using our optimization goals as “guides”
    - Or more formally - constraints.
- We know where we are currently, we know where we’d like to be - given this, how do we compute H_t

The Go GC Pacer made use of a *proportional controller* for this.

GC Pacer Prior To Go 1.18

- How does the controller work?

GC Pacer Prior To Go 1.18

- How does the controller work?
  - We could just adjust the trigger point based on how much the heap over or under-shot the goal.

GC Pacer Prior To Go 1.18

- How does the controller work?
  - We could just adjust the trigger point based on how much the heap over or under-shot the goal.
    - H_g `- sizeOfHeapEndOfCycle`

GC Pacer Prior To Go 1.18

- How does the controller work?
  - We could just adjust the trigger point based on how much the heap over or under-shot the goal.
    - H_g `- sizeOfHeapEndOfCycle`
    - But this does not take into account our CPU utilization goal
    - So, instead we ask the following:

GC Pacer Prior To Go 1.18

- How does the controller work?
  - Assuming that we are at goal utilization, how much *would* the heap have grown since last cycle?

GC Pacer Prior To Go 1.18

- How does the controller work?
  - Assuming that we are at goal utilization, how much *would* the heap have grown since last cycle?
  - If we are at double the utilization:
    - This is probably because we do double the scan work (through dedicated mark workers or assists)

GC Pacer Prior To Go 1.18

- How does the controller work?
  - Assuming that we are at goal utilization, how much *would* the heap have grown since last cycle?
  - If we are at double the utilization:
    - This is probably because we do double the scan work (through dedicated mark workers or assists)
    - Which implies the heap grew to twice the size it was expected to (heap goal).

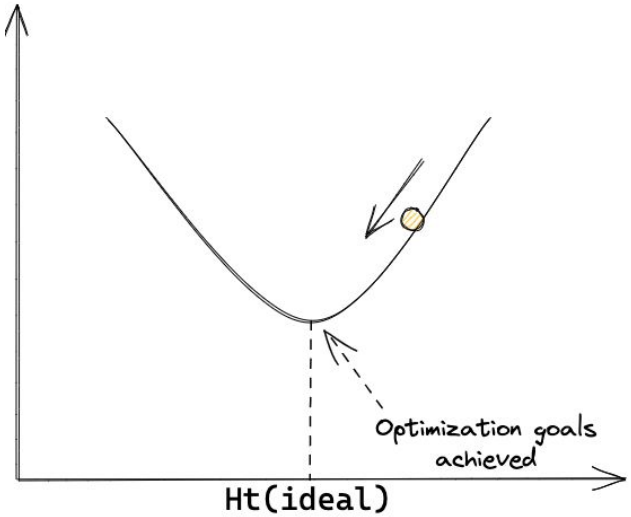
GC Pacer Prior To Go 1.18

- How does the controller work?
  - Assuming that we are at goal utilization, how much *would* the heap have grown since last cycle?
  - If we are at double the utilization:
    - This is probably because we do double the scan work (through dedicated mark workers or assists)
    - Which implies the heap grew to twice the size it was expected to (heap goal).
    - **Which means we should try and start the cycle *earlier* next time.**

We are essentially trying to determine a point such that we optimize our 2 goals.

GC Pacer Prior To Go 1.18

We are essentially trying to determine a point such that we optimize our 2 goals.



GC Pacer Prior To Go 1.18

- How does the controller work?
  - If the heap does end up overshooting:
    - There should be a maximum amount by which this should happen
    - This is defined as the “hard” goal

GC Pacer Prior To Go 1.18

- How does the controller work?
  - If the heap does end up overshooting:
    - There should be a maximum amount by which this should happen
    - This is defined as the “hard” goal

The hard goal is defined as 1.1 times the heap goal.

GC Pacer Prior To Go 1.18

Let’s Talk About Assists

GC Pacer Prior To Go 1.18

- How does the controller work?
  - Ideally, in the steady state - we should not have any mark assists.

GC Pacer Prior To Go 1.18

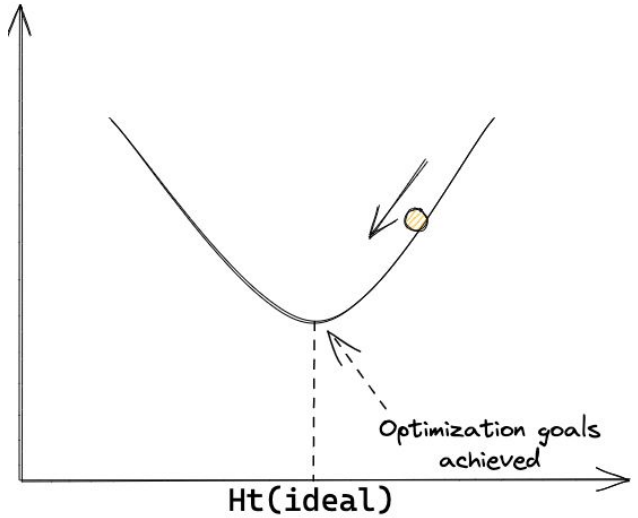
- How does the controller work?
  - Ideally, in the steady state - we should not have any mark assists.
  - Due to the way the error term of the P controller is, it can go to 0 even when our optimization goals are not met
    - If this persists, this can trick the controller into thinking that all's good - because look! No error!

GC Pacer Prior To Go 1.18

- How does the controller work?
  - Ideally, in the steady state - we should not have any mark assists.
  - Due to the way the error term of the P controller is, it can go to 0 even when our optimization goals are not met
    - If this persists, this can trick the controller into thinking that all's good - because look! No error!

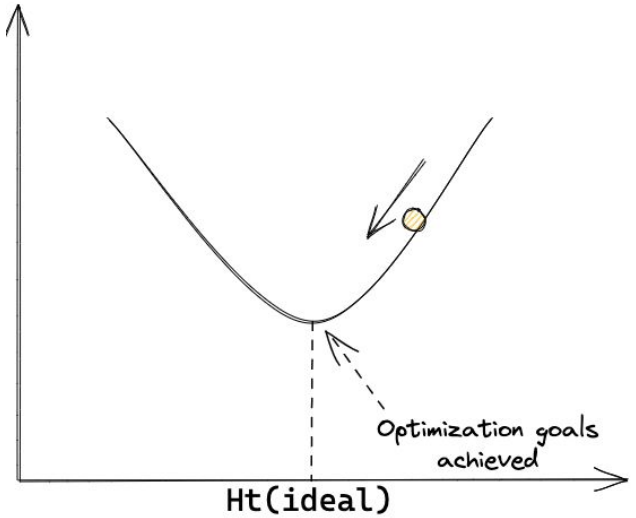
GC Pacer Prior To Go 1.18

- How does the controller work?
  - If the error is 0 due to goals being met or not means asking ” are we under pacing, over pacing or pacing on point?”



GC Pacer Prior To Go 1.18

- How does the controller work?
  - If the error is 0 due to goals being met or not means asking ” are we under pacing, over pacing or pacing on point?”
  - To know the answer to this question, we need to actually perform GC assists.
    - This is where the 5% extension comes from!



GC Pacer Prior To Go 1.18

So, that sounds great and everything, but what's the downside?

GC Pacer Prior To Go 1.18

Downside 1: When non-heap sources of work are not negligible.

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

- When GOGC is large, there is a lot of runway in terms of how much the heap can grow.

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

- When GOGC is large, there is a lot of runway in terms of how much the heap can grow.
- If at some point during the cycle, all of the memory turns out to be live:

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

- When GOGC is large, there is a lot of runway in terms of how much the heap can grow.
- If at some point during the cycle, all of the memory turns out to be live:
    - And we have the hard heap goal to adhere to:

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

- When GOGC is large, there is a lot of runway in terms of how much the heap can grow.
- If at some point during the cycle, all of the memory turns out to be live:
    - And we have the hard heap goal to adhere to:
        - To try and meet it, rate of assists will skyrocket, starving mutators.

GC Pacer Prior To Go 1.18

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

- When GOGC is large, there is a lot of runway in terms of how much the heap can grow.
- If at some point during the cycle, all of the memory turns out to be live:
    - And we have the hard heap goal to adhere to:
        - To try and meet it, rate of assists will skyrocket, starving mutators.
            - And recovering from this itself can take a while!

GC Pacer Prior To Go 1.18

Downside 3: The steady state error of a P Controller will never converge to 0.

GC Pacer Prior To Go 1.18

Downside 4 : Mark assists are part of the steady state (5% extension from the 25% goal)

GC Pacer Prior To Go 1.18

Downside 1: When non-heap sources of work are not negligible.

Downside 2: When GOGC is large, changes to live heap size causes excessive assists.

Downside 3: The steady state error of a P Controller will never converge to 0.

Downside 4 : Mark assists elimination in the steady state (30%).

The GC Pacer Redesign!

GC Pacer Since Go 1.18

GC Pacer Since Go 1.18

Major Trends:

GC Pacer Since Go 1.18

Major Trends:

1. Include non-heap sources of work in pacing decisions.

GC Pacer Since Go 1.18

Major Trends:

1. Include non-heap sources of work in pacing decisions.
2. Re-frame the pacing decision as a “search problem”.

GC Pacer Since Go 1.18

Major Trends:

1. Include non-heap sources of work in pacing decisions.
2. Re-frame the pacing decision as a “search problem”
3. Use a PI Controller.

GC Pacer Since Go 1.18

Major Trends:

1. Include non-heap sources of work in pacing decisions.
2. Re-frame the pacing decision as a “search problem”
3. Use a PI Controller
4. Change target CPU utilization to 25%.

GC Pacer Since Go 1.18

Include Non-Heap Sources of Work In Pacing Decisions

GC Pacer Since Go 1.18

Include Non-Heap Sources of Work In Pacing Decisions

- Previously, we just considered heap as the source of GC work.
- Now, we also include non heap sources of work, namely stacks and globals.

GC Pacer Since Go 1.18

Include Non-Heap Sources of Work In Pacing Decisions

- Previously, we just considered heap as the source of GC work.
- Now, we also include non heap sources of work, namely stacks and globals.

$$H_g(n) = [H_m(n-1) + S_n + G_n] \times [1 + GOGC/100]$$

GC Pacer Since Go 1.18

Include Non-Heap Sources of Work In Pacing Decisions

$$H_g(n) = [H_m(n-1) + S_n + G_n] \times [1 + GOGC/100]$$

- This now essentially changes the expected behaviour of GOGC!

GC Pacer Since Go 1.18

Include Non-Heap Sources of Work In Pacing Decisions

$$H_g(n) = [H_m(n-1) + S_n + G_n] \times [1 + GOGC/100]$$

- This now essentially changes the expected behaviour of GOGC!
- Most programs with default GOGC values, are likely to now end up using more memory.

GC Pacer Since Go 1.18

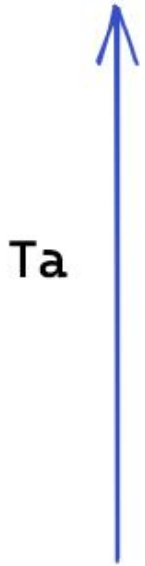
Let’s take a step back for a minute

GC Pacer Since Go 1.18

The GC Pacer Knows 2 ***Notions*** of time:

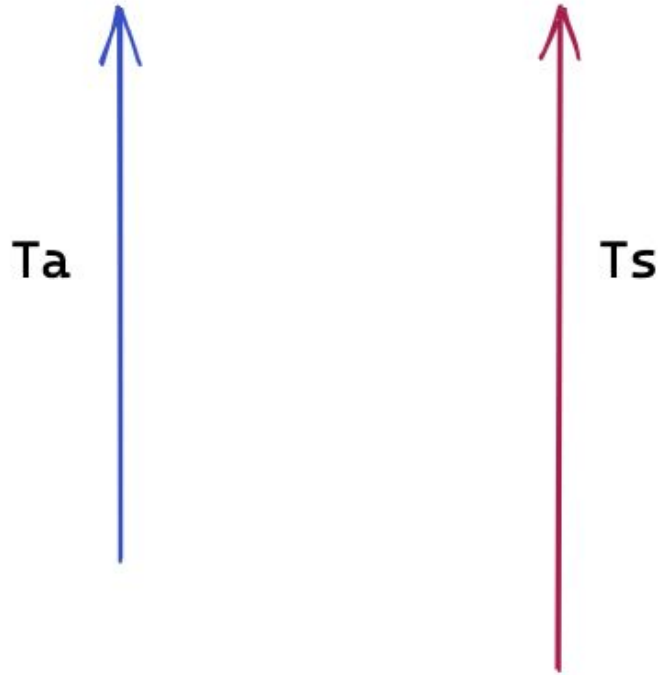
GC Pacer Since Go 1.18

- T_a - time taken to allocate after trigger



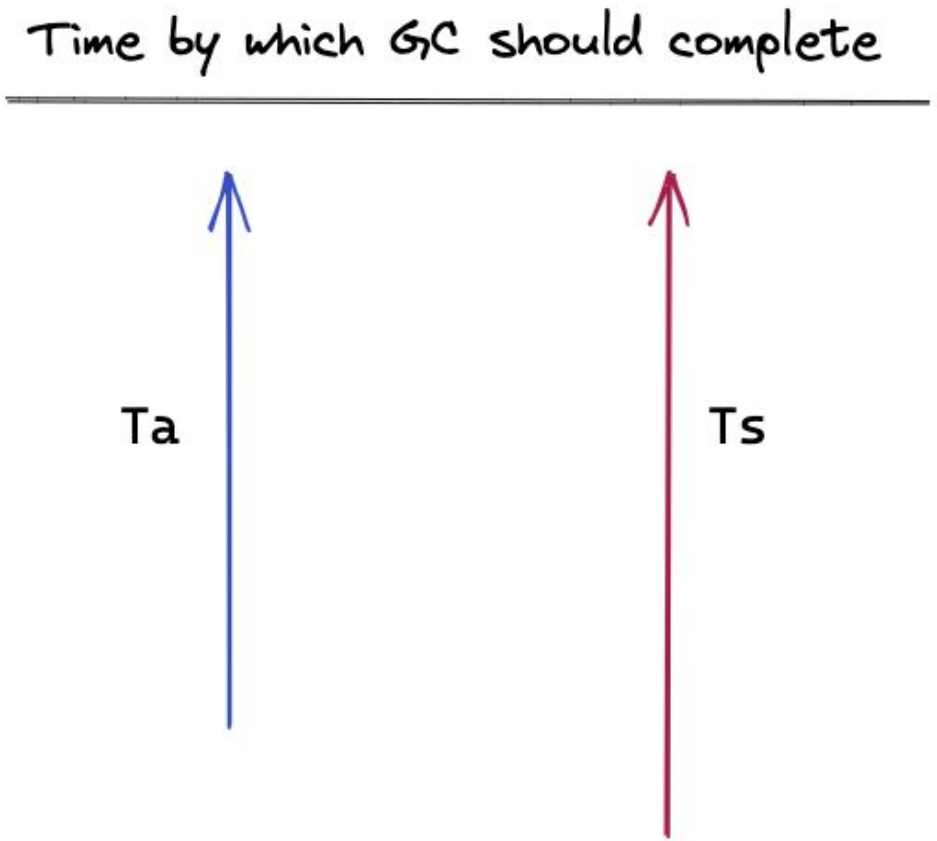
GC Pacer Since Go 1.18

- T_a - time taken to allocate after trigger.
- T_s - time taken to perform GC work.



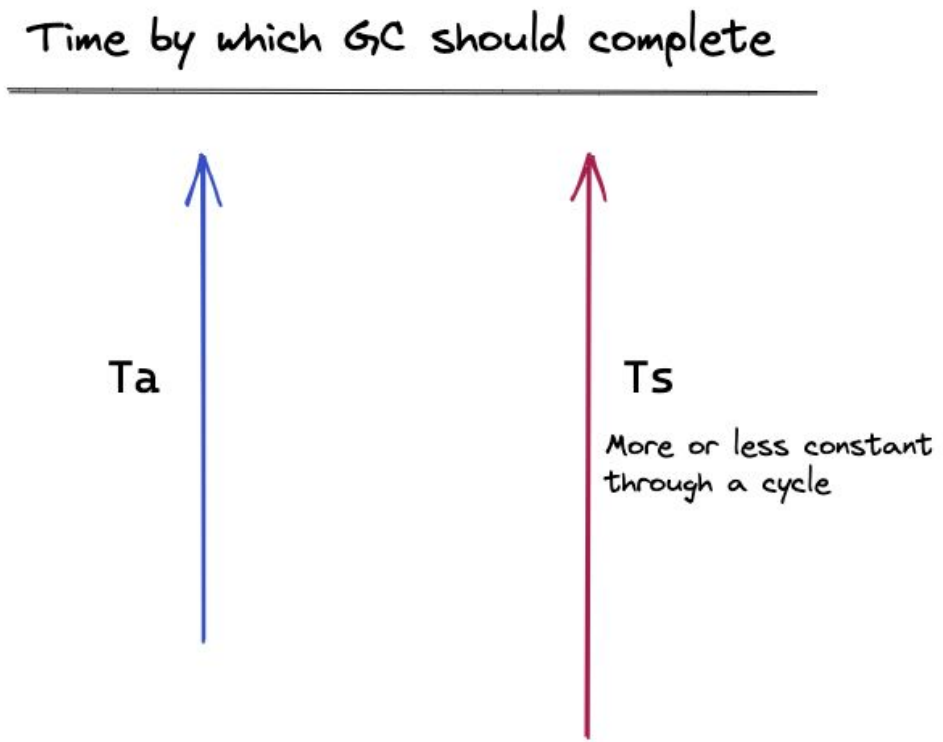
GC Pacer Since Go 1.18

- T_a - time taken to allocate after trigger.
- T_s - time taken to perform GC work.
- Ideally, the pacer needs to “complete” these in the same amount.



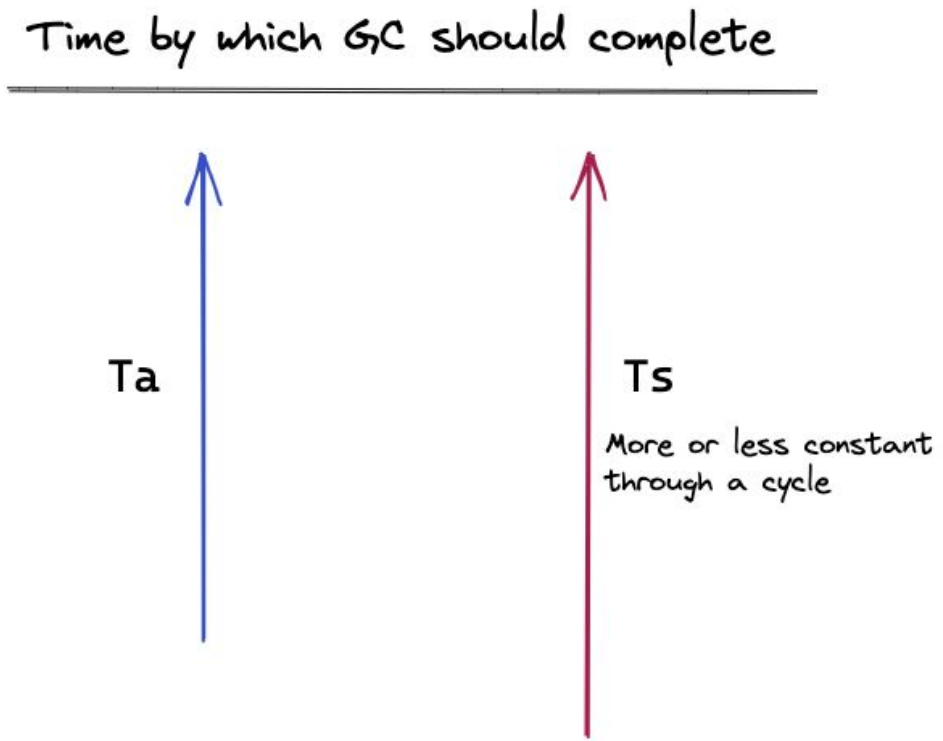
GC Pacer Since Go 1.18

- T_a - time taken to allocate after trigger.
- T_s - time taken to perform GC work.
- Ideally, the pacer needs to “complete” these in the same amount.
- In the steady state, the amount of GC work is roughly going to be constant.



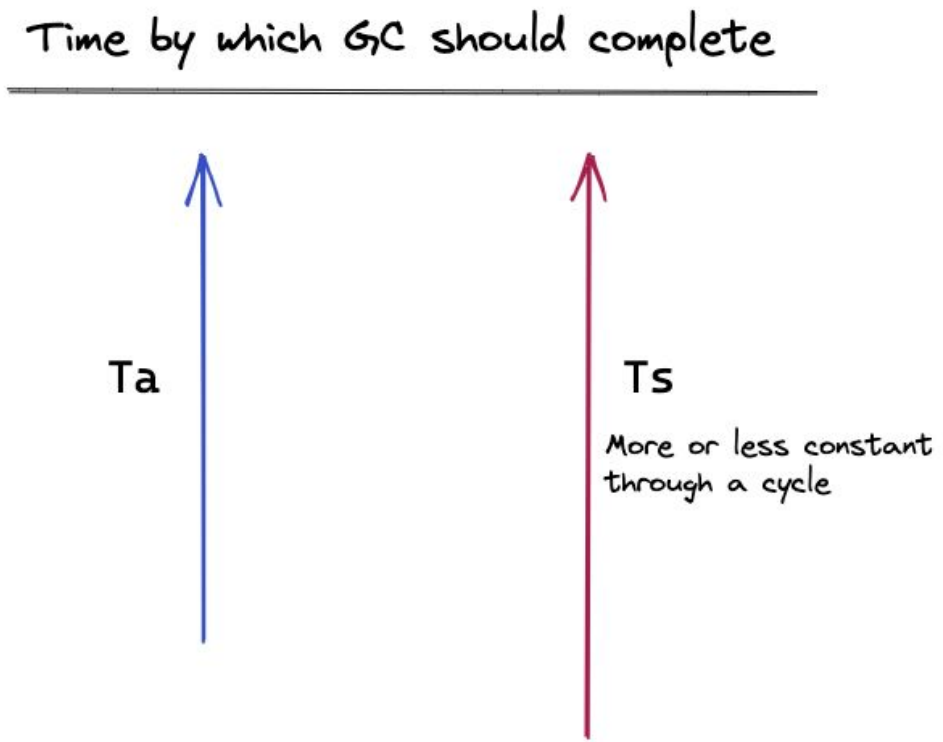
GC Pacer Since Go 1.18

- Continuing from this:
  - Our application can either spend its time on itself (could be allocating too) or doing GC work.



GC Pacer Since Go 1.18

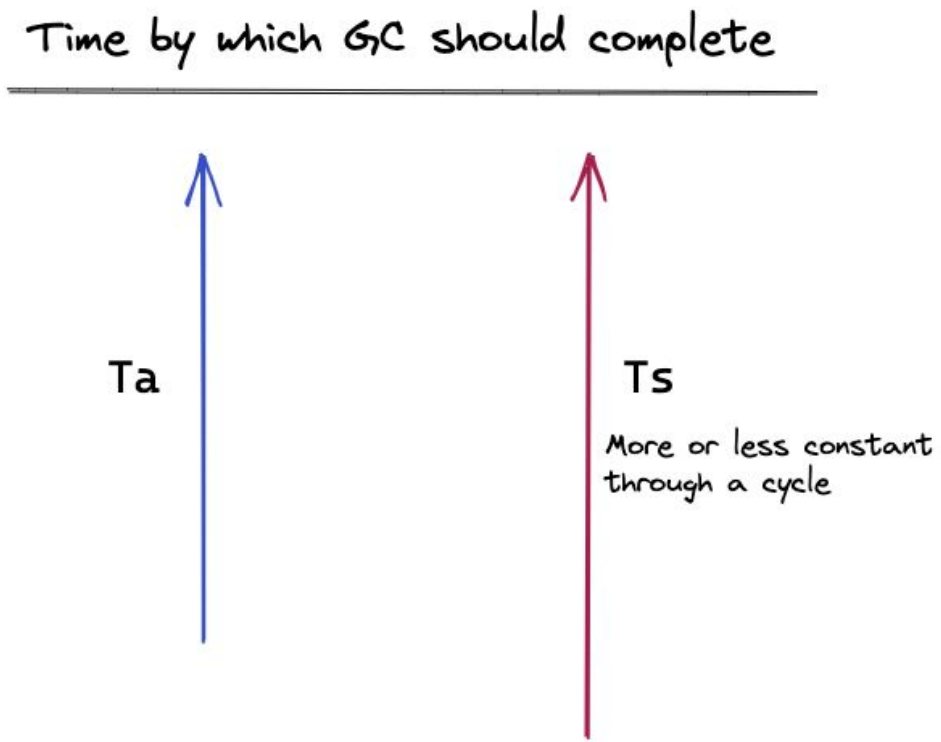
- Continuing from this:
  - Our application can either spend its time on itself (could be allocating too) or doing GC work.
  - So, these 2 notions of time could be thought of as “bytes allocated” and “bytes scanned”.



GC Pacer Since Go 1.18

- Considering these 2 notions of time need to “complete” at the same time …
- If B_a is bytes allocated and B_s is bytes scanned, the amount we scan is going to be proportional to the amount we allocate, which means:

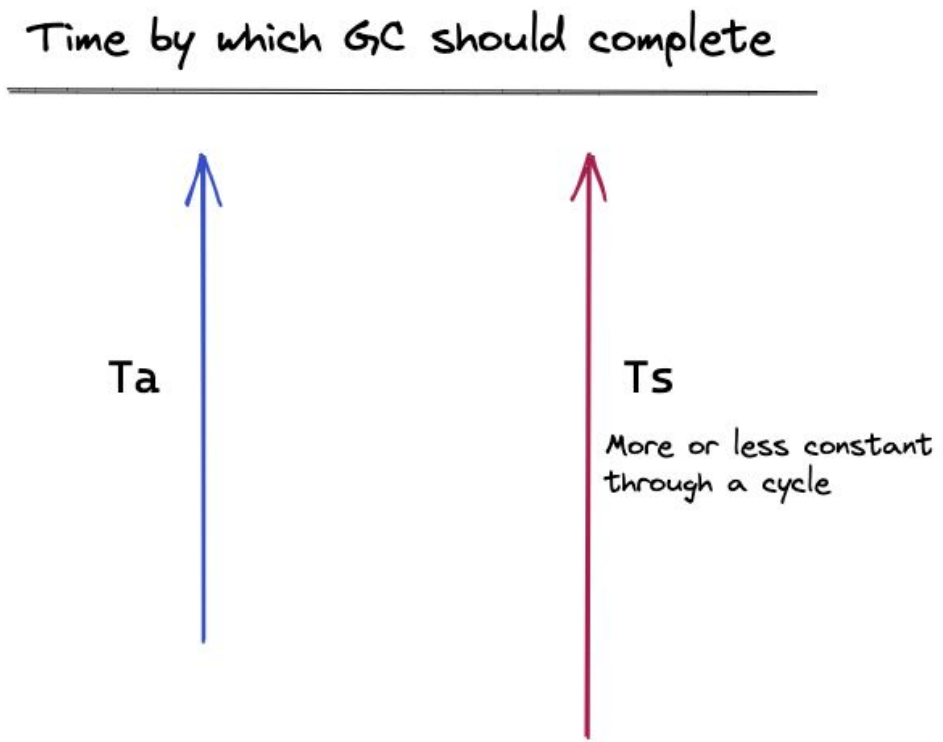
$$B_a = someConstant \times B_s$$



GC Pacer Since Go 1.18

- Considering these 2 notions of time need to “complete” at the same time …
- If B_a is bytes allocated and B_s is bytes scanned, the amount we scan is going to be proportional to the amount we allocate, which means:

$$B_a = r \times B_s$$

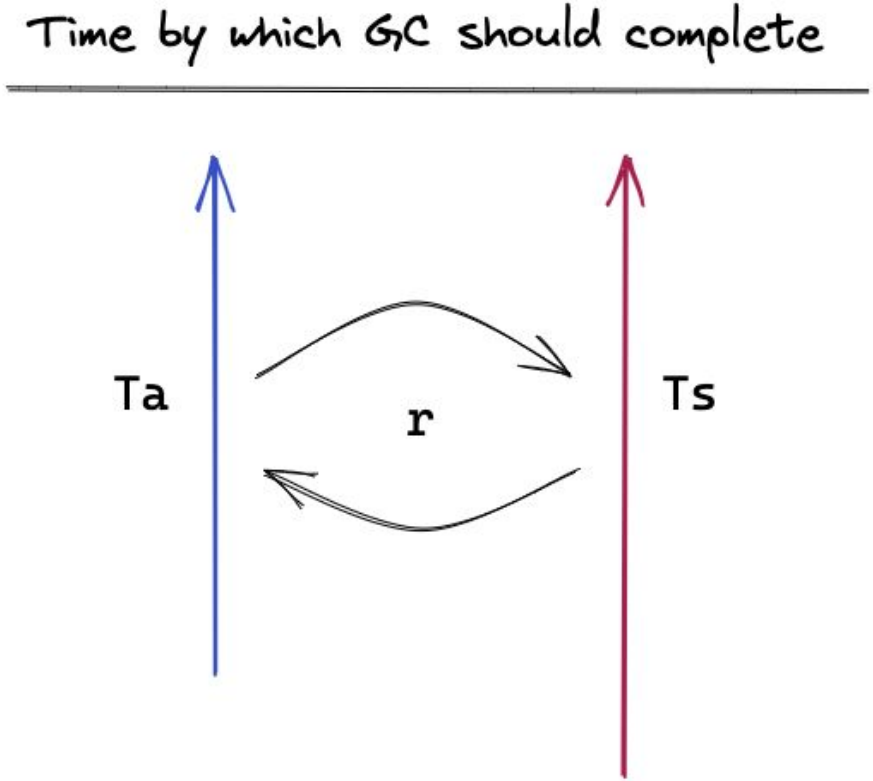


GC Pacer Since Go 1.18

- Considering these 2 notions of time need to “complete” at the same time …
- If B_a is bytes allocated and B_s is bytes scanned (heap, stacks, globals included):

$$r = B_a / B_s$$

- This acts as a ***conversion*** factor between these 2 notions of time.



GC Pacer Since Go 1.18

$$r = B_a / B_s$$

- Subsequently, we'd like these 2 notions of time to "complete" in the same amount while maintaining the target CPU utilization.
- To achieve this, we scale `r`:

$$r = [B_a / B_s] \times K(u_T, u_n)$$

GC Pacer Since Go 1.18

- If we know what our goal is, and we know somehow know how many bytes will be allocated in a GC cycle, we can reliably calculate *when* to start a GC cycle.

GC Pacer Since Go 1.18

- If we know what our goal is, and we know somehow know how many bytes will be allocated in a GC cycle, we can reliably calculate *when* to start a GC cycle.

$$T_n = H_g - rB_s$$

GC Pacer Since Go 1.18

$$T_n = H_g - rB_s$$

- Intuitively, the size of the live heap (A) when we start a GC cycle, will always be greater than (or in some extreme cases, equal to) T_n

$$A \geq T_n$$

GC Pacer Since Go 1.18

$$T_n = H_g - rB_s$$

- Intuitively, the size of the live heap (A) when we start a GC cycle, will always be greater than (or in some extreme cases, equal to) T_n

$$A \geq T_n$$

$$\Rightarrow A \geq H_g - rB_s$$

GC Pacer Since Go 1.18

$$A \geq H_g - rB_s$$

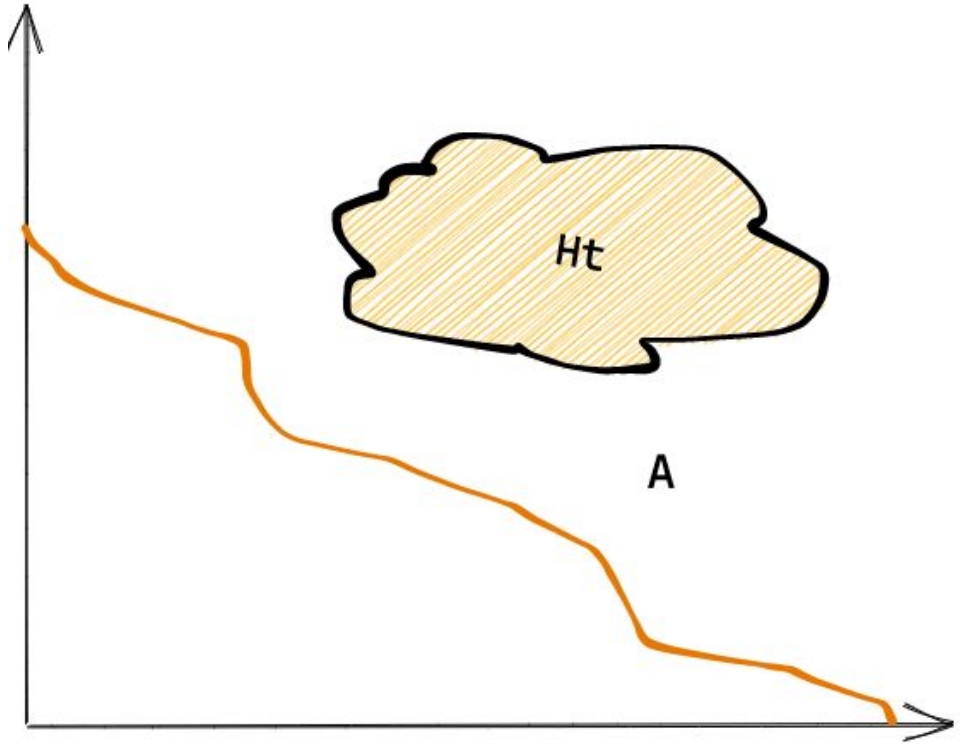
- This is a *condition* not a predetermined trigger point as before.
- We now have a *search space* formulated by a condition that encapsulates both our optimization goals!

GC Pacer Since Go 1.18

$$A \geq H_g - rB_s$$

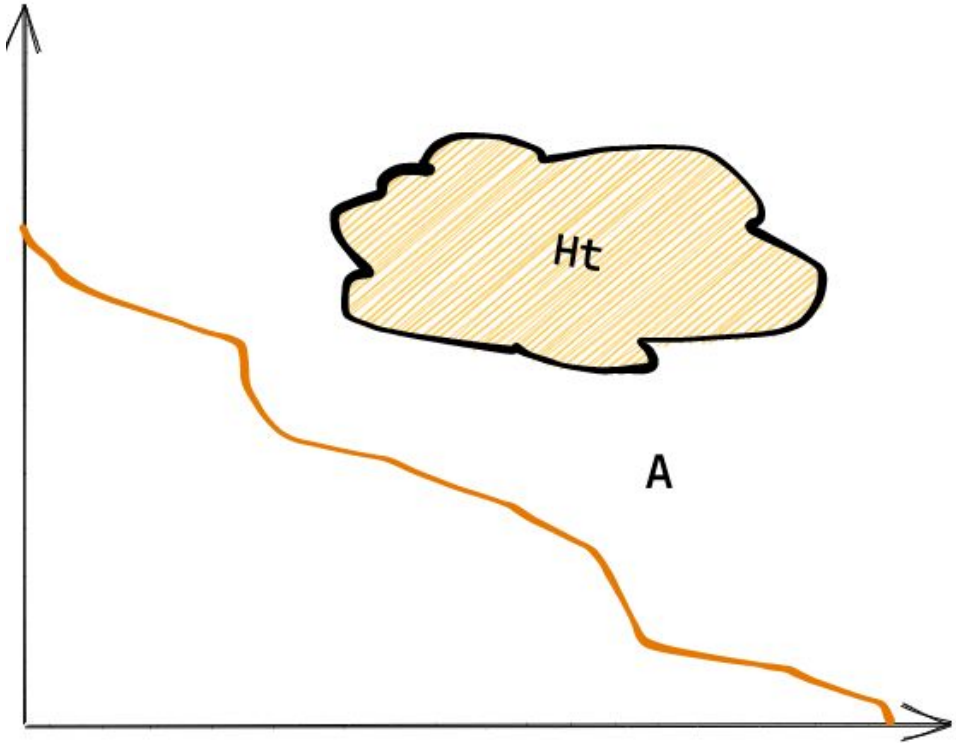
GC Pacer Since Go 1.18

$$A \geq H_g - rB_s$$

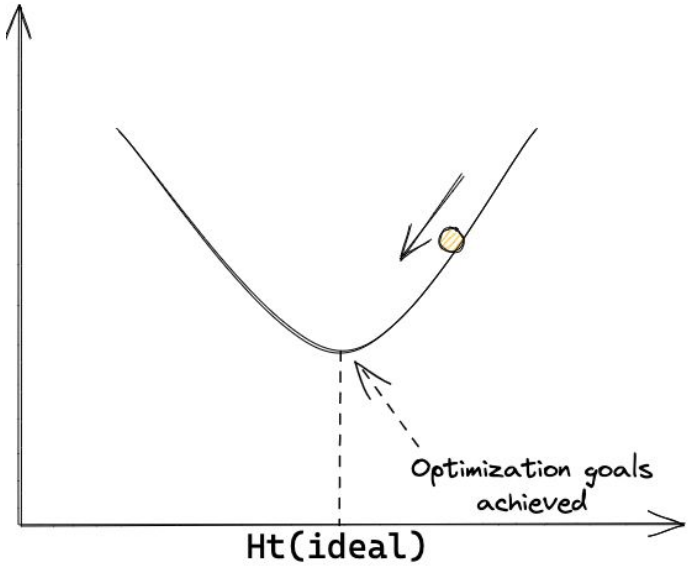


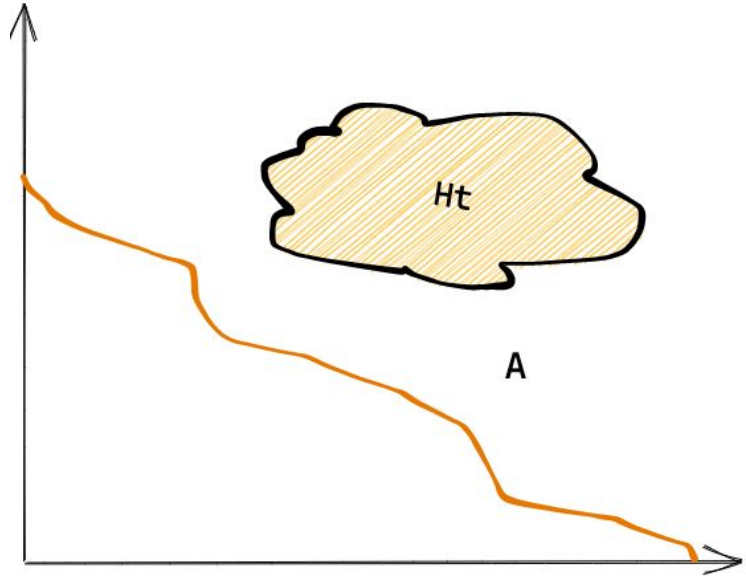
GC Pacer Since Go 1.18

- Since we know H_g and the amount of scan work, we need to *search* for a value of `r` such that we trigger at the right point.
- This converts our pacing problem into a search problem from an optimization one.



GC Pacer Since Go 1.18





GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

GC Pacer Since Go 1.18

How do we calculate **r** over a GC cycle?

$$r = [B_a / B_s] \times K(u_T, u_n)$$

- At the end of a GC cycle, B_a is **PeakLiveHeap - Trigger**
  - Or in other words - the amount we have allocated since the cycle started.

GC Pacer Since Go 1.18

How do we calculate **`r`** over a GC cycle?

$$r = [B_a / B_s] \times K(u_T, u_n)$$

- At the end of a GC cycle, B_a is **`PeakLiveHeap - Trigger`**
  - Or in other words - the amount we have allocated since the cycle started.
- This value can be calculated only at the end of the cycle, and it is what the value of **`r` should have** been in order to meet our target.

GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

$$r = [B_a / B_s] \times K(u_T, u_n)$$

- At the end of a GC cycle, B_a is `PeakLiveHeap - Trigger`
  - Or in other words - the amount we have allocated since the cycle started.
- This value can be calculated only at the end of the cycle, and it is what the value of **`r` should have** been in order to meet our target.
- In the steady state, we would expect the next GC cycle to also be similar to this one, if that is true, it stands to reason that we can use this value of `r` for the next cycle.

GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

- Turns out using this value directly is very noisy! And this might end up missing the target.

GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

- Turns out using this value directly is very noisy! And this might end up missing the target.
- Instead, we aspire to search for a more “ideal” r value in the long run and use a **PI controller** as our way to try and search for it by setting the calculated r value as discussed to be the set point.

GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

- Turns out using this value directly is very noisy! And this might end up missing the target.
- Instead, we aspire to search for a more “ideal” r value in the long run and use a **PI controller** as our way to try and search for it by setting the calculated r value as discussed to be the set point.
- The controller might bounce around a little bit but the value it bounces around will probably be a better r value than what we would have used.

GC Pacer Since Go 1.18

How do we calculate `r` over a GC cycle?

- Turns out using this value directly is very noisy! And this might end up missing the target.
- Instead, we aspire to search for a more “ideal” r value in the long run and use a **PI controller** as our way to try and search for it by setting the calculated r value as discussed to be the set point.
- The controller might bounce around a little bit but the value it bounces around will probably be a better r value than what we would have used.

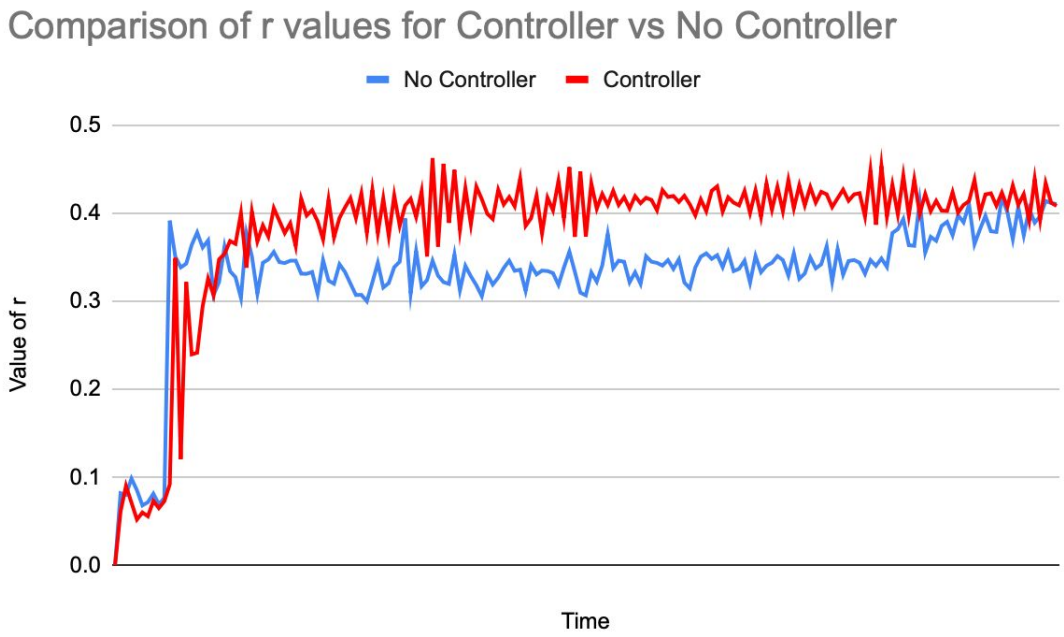
What does that look like?

GC Pacer Since Go 1.18

```
GODEBUG=gcpacertrace=1
```

- This outputs a handful of metrics internal to the tracer, such as the `r` value, amount of scan work to be done etc.

GC Pacer Since Go 1.18



GC Pacer Since Go 1.18

- Due to this way of doing things:
  - We reframe our pacing problem and no longer suffer from the issue of controller getting saturated due to the P-only error term.

GC Pacer Since Go 1.18

- Due to this way of doing things:
  - We reframe our pacing problem and no longer suffer from the issue of controller getting saturated due to the P-only error term.
  - Which means we no longer need mark assists in the steady state

GC Pacer Since Go 1.18

- Due to this way of doing things:
  - We reframe our pacing problem and no longer suffer from the issue of controller getting saturated due to the P-only error term.
  - Which means we no longer need mark assists in the steady state
  - And don’t require the 5% extension - *the goal utilization can be reduced to 25%!*

GC Pacer Since Go 1.18

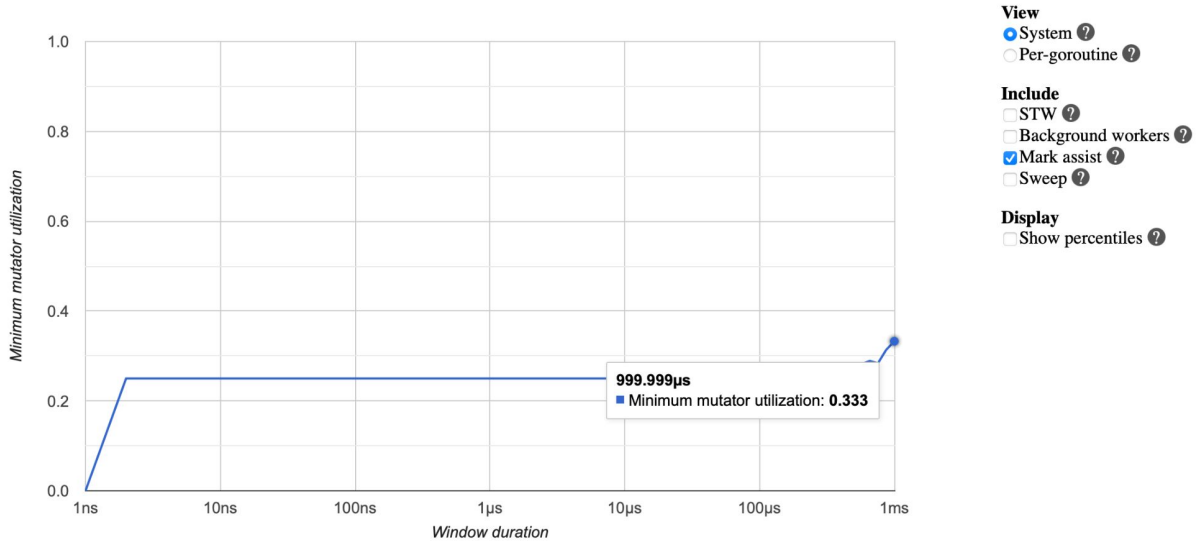
- Due to this way of doing things:
  - We reframe our pacing problem and no longer suffer from the issue of controller getting saturated due to the P-only error term.
  - Which means we no longer need mark assists in the steady state
  - And don’t require the 5% extension - *the goal utilization can be reduced to 25%!*
  - This potentially means better application latencies as well!

GC Pacer Since Go 1.18

If we run the `garbage` benchmark: `garbage -benchtime 30s -benchmem 512` and collect execution traces

For Go 1.17, the minimum

mutator utilization curve:

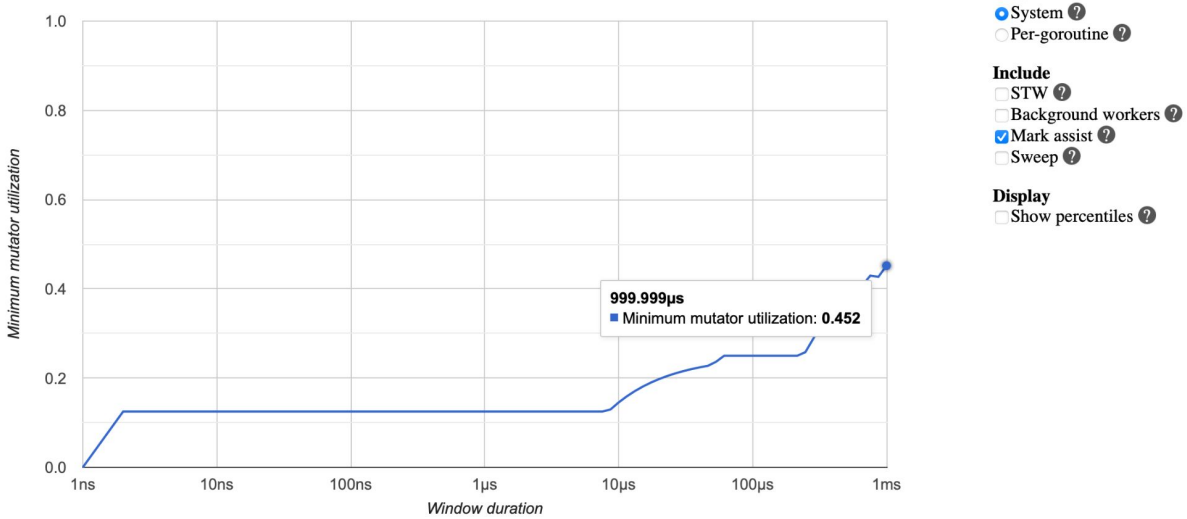


GC Pacer Since Go 1.18

If we run the `garbage` benchmark: `garbage -benchtime 30s -benchmem 512` and collect execution traces

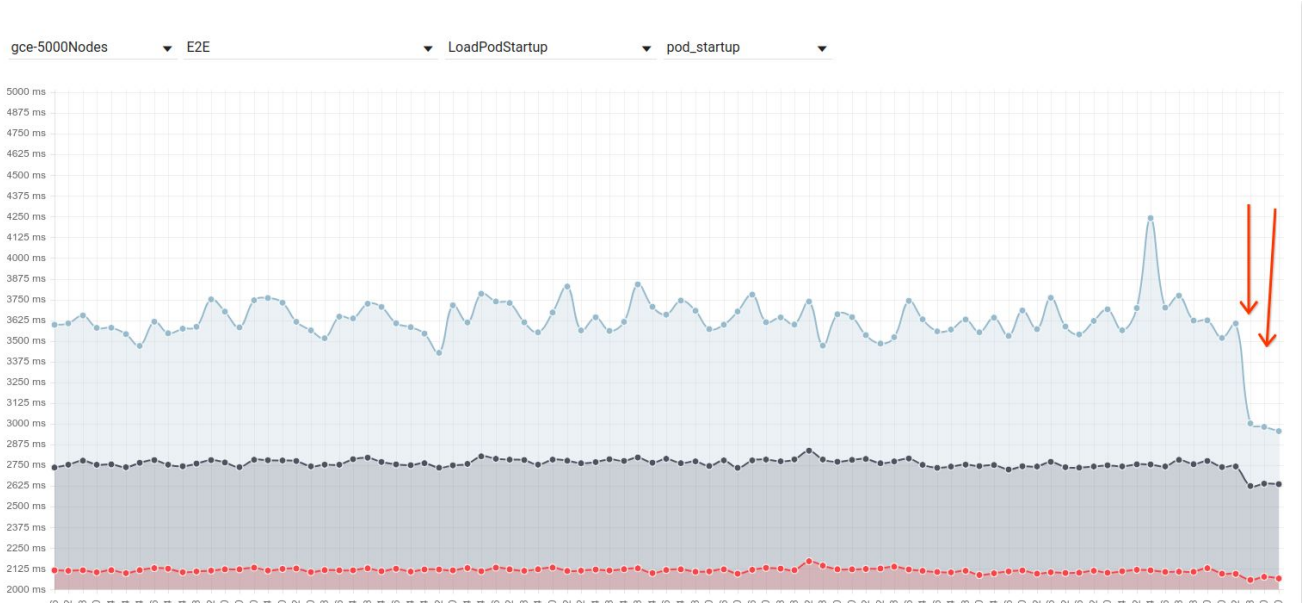
For Go 1.18, the minimum

mutator utilization curve:



GC Pacer Since Go 1.18

On scalability tests run on 5000 node Kubernetes clusters, Pod Startup Latency seemed to improve significantly after shifting to Go 1.18 (Thank you Antoni and Marseel from SIG Scalability for helping out with this picture!)



GC Pacer Since Go 1.18

Okay, phew… so, till now in our discussion of the redesign, we've spoken about:

- Including non-heap sources of work in pacing decisions
- Reframing the problem as a search problem
- Making use of a PI Controller for this search problem

GC Pacer Since Go 1.18

So far, this helps us mitigate the following downsides that we previously had:

- P-only controller disadvantages.
- Cases where non-heap sources of work are significant.
- We reduce the goal utilization to 25%, potentially improving application latencies.

GC Pacer Since Go 1.18

Okay, but what about assists?

GC Pacer Since Go 1.18

- Assists come into play when we find more GC work than expected (non-heap sources included).

GC Pacer Since Go 1.18

- Assists come into play when we find more GC work than expected (non-heap sources included).
- The worst case is when all scannable memory turns out to be live.

GC Pacer Since Go 1.18

- Assists come into play when we find more GC work than expected (non-heap sources included).
- The worst case is when all scannable memory turns out to be live.
- Previously, we always assumed that the worst case is likely to happen, and bounded the heap growth to 1.1x of the heap goal (arbitrarily).

GC Pacer Since Go 1.18

- Assists come into play when we find more GC work than expected (non-heap sources included).
- The worst case is when all scannable memory turns out to be live.
- Previously, we always assumed that the worst case is likely to happen, and bounded the heap growth to 1.1x of the heap goal (arbitrarily).
    - But, when we overshoot this hard limit (in cases where we have a large GOGC and we have the runway to do so) and the worst case actually happens, the rate of assists would skyrocket, starving mutators.

GC Pacer Since Go 1.18

- But, if we have all this live memory, the next GC cycle is going to use *at least* this much memory anyway!

GC Pacer Since Go 1.18

- But, if we have all this live memory, the next GC cycle is going to use *at least* this much memory anyway!
- So, why panic and ramp up assists, let's let it slide for now and keep our rate of assists calm and smooth.

GC Pacer Since Go 1.18

- But, if we have all this live memory, the next GC cycle is going to use *at least* this much memory anyway!
- So, why panic and ramp up assists, let's let it slide for now and keep our rate of assists calm and smooth.
- But we cannot let this "deferring" shoot up the heap goal of the next cycle either.

GC Pacer Since Go 1.18

- But, if we have all this live memory, the next GC cycle is going to use *at least* this much memory anyway!
- So, why panic and ramp up assists, let's let it slide for now and keep our rate of assists calm and smooth.
- But we cannot let this "deferring" shoot up the heap goal of the next cycle either.
- Let $\mathtt{H}_{\mathtt{L}}$ be the size of the original live heap.

GC Pacer Since Go 1.18

- Let H_L be the size of the original live heap.
- In steady state, heap goal for the current cycle would be:

$$[1 + GOGC/100] \times H_L$$

- And the heap goal for the next cycle would be:

$$[1 + GOGC/100] \times [1 + GOGC/100] \times H_L$$

GC Pacer Since Go 1.18

$$[1 + GOGC/100] \times [1 + GOGC/100] \times H_L$$

- Assuming GOGC = 100, the worst case memory usage of next cycle would be 4x the size of the original live heap.
- Maintaining this invariant, we now extend the hard heap goal of this cycle to the worst case heap goal of the next cycle.
- Allow using more memory in the current cycle, because the next cycle is going to use at least this much extra memory anyway.

GC Pacer Since Go 1.18

$$[1 + GOGC/100] \times [1 + GOGC/100] \times H_L$$

- This shields us from skyrocketing mark assist rates, but in the worst case, program memory consumption could spike up to 4x (for GOGC = 100) of the original live heap.

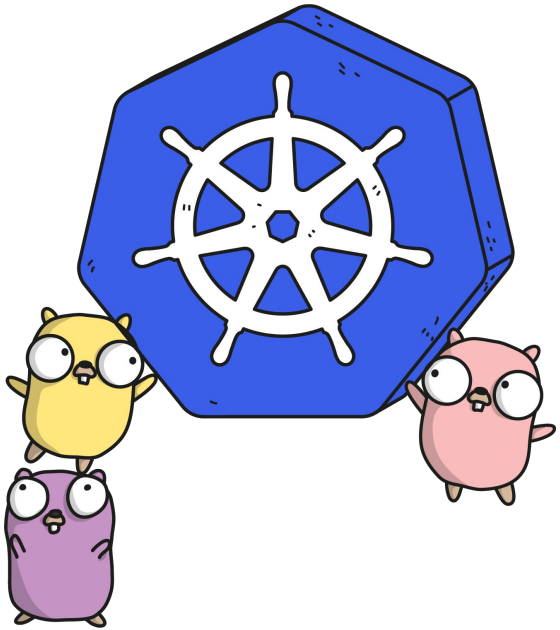
GC Pacer Since Go 1.18

$$[1 + GOGC/100] \times [1 + GOGC/100] \times H_L$$

- *This shields us from skyrocketing mark assist rates, but in the worst case, program memory consumption could spike up to 4x (for GOGC = 100) of the original live heap.*
- For the sake of robustness, in some truly worst case scenarios, we bound this scenario also to 1.1x of the worst case goal.
    - So, in these scenarios, program memory *could* spike up to

$$1.1 \times [1 + GOGC/100] \times [1 + GOGC/100] \times H_L$$

How Did These Changes Affect A Kubernetes Release?



How Did These Changes Affect A Kubernetes Release?

- Kubernetes runs scalability tests on clusters of different sizes (100, 500, 5000 nodes).
- When the change to Go 1.18 was made:

How Did These Changes Affect A Kubernetes Release?

- Kubernetes runs scalability tests on clusters of different sizes (100, 500, 5000 nodes).
- When the change to Go 1.18 was made:
    - All clusters experienced a noticeable increase in memory consumption.

How Did These Changes Affect A Kubernetes Release?

- Kubernetes runs scalability tests on clusters of different sizes (100, 500, 5000 nodes).
- When the change to Go 1.18 was made:
  - All clusters experienced a noticeable increase in memory consumption.
  - Tests running on 5000 nodes experienced a sharp drop in Pod Startup Latencies
    - Pod Startup Latency = time taken since pod was created to when all its containers are reported as started when observed via a watch.
- The learnings from this are generally applicable to all Go programs as well and not specific to just Kubernetes!

How Did These Changes Affect A Kubernetes Release?

- In the 5000 node cluster, the memory footprint of the kube-apiserver increased by at least 10%, causing a release blocking scalability regression.
- This increased footprint was due to the pacer redesign, specifically, due to change in the meaning of `GOGC`.

How Did These Changes Affect A Kubernetes Release?

- In the 5000 node cluster, the memory footprint of the kube-apiserver increased by at least 10%, causing a release blocking scalability regression.
- This increased footprint was due to the pacer redesign, specifically, due to change in the meaning of `GOGC`.
- The peak live heap size tries to be as close to the heap goal of that cycle (the pacer tries to do this)
    - Previously, the heap goal would only factor in heap sources of work.
    - Now, it also considers stacks and globals, therefore increasing the heap goal by some amount.

How Did These Changes Affect A Kubernetes Release?

- In the 5000 node cluster, the memory footprint of the kube-apiserver increased by at least 10%, causing a release blocking scalability regression.
- This increased footprint was due to the pacer redesign, specifically, due to change in the meaning of `GOGC`.
- ***A Go program experiences a noticeable increase in memory usage after switching to Go 1.18 if it has non-negligible amounts of non-heap memory compared to heap memory.***

How Did These Changes Affect A Kubernetes Release?

The solution to mitigate this is to adjust GOGC accordingly!

How Did These Changes Affect A Kubernetes Release?

The solution to mitigate this is to adjust GOGC accordingly!

Note: Kubernetes *does not* set GOGC or any other runtime variables.

Mitigating Increased Memory Usage Due To Pacer Redesign

By adjusting GOGC, we can return to our previous memory consumption.

But what do we set GOGC to?

Mitigating Increased Memory Usage Due To Pacer Redesign

- If M_o is the old memory consumption and M_n is the new increased memory consumption, an approximation of the new GOGC can be:

$$M_o \times [1 + GOGC_{old} / 100] = M_n \times [1 + GOGC_{new} / 100]$$

- And then we solve for $GOGC_{new}$

Mitigating Increased Memory Usage Due To Pacer Redesign

$$M_o \times [1 + GOGC_{old} / 100] = M_n \times [1 + GOGC_{new} / 100]$$

- M_n can also be derived from M_o if we know how much heap and non-heap memory we are using after making the switch to Go 1.18 or higher.

$$M_n = [1 + non\text{-}heap/heap] \times M_o$$

A Note On Go 1.19 Soft Memory Limit

A Note On Go 1.19 Soft Memory Limit

- Go 1.19 introduced a limit on the total amount of memory the Go runtime can use (to help mitigate cases of out of memory errors and GC workarounds).
- The pacer ties in tightly with this limit since it has to decide when to start a GC cycle (now also trying to respect this limit).
- The pacer tries to set the heap goal as the minimum of our previous definition and the heap limit (that is derived from `GOMEMLIMIT`).
- This change also limits the GC CPU consumption to 50% (compromising meeting the heap goal in some cases), and post this, the GC gives back time to the application in order to prevent “death spirals”.

References

- GC Pacer Redesign Design Proposal
- Original GC Pacer Design Proposal
- golang/go#42430 (GC Pacer Meta Issue)
- golang/go#14951 (tracking aggressive assist rates)
- Separate Soft and Hard Heap Limit Design Proposal
- On `release-branch.go.1.18`
  - `src/runtime/{mgc.go,mgcpacer.go,proc.go}`
- Pod Startup Latency SLO
- Kubernetes Performance Dashboard
- A Guide to the Go Garbage Collector
- Commit Message of Change Implementing The Redesign
- A Parallel, Real-Time Garbage Collector
- Garbage Collection Semantics
- Go GC: Solving The Latency Problem
- Loop Preemption in Go 1.14
- Golang Garbage Collection Benchmarks
- Introduction to Control Theory And Its Applications to Computing Systems
- Control Theory in Container Fleet Management
- Feedback Control for Computer Systems

Thank you!

Twitter: @MadhavJivrajani

Gophers/K8s/CNCF Slack: @madhav