Cache Contention Aware Go Scheduler

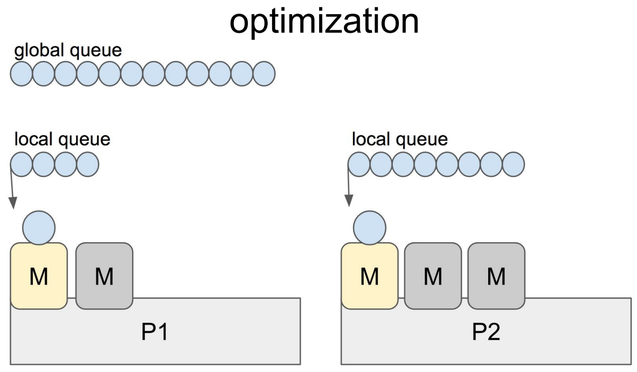
Craig Hesling and Sannan Tariq

Background

Go has been gaining popularity recently, backed by its heavy use in Google products due to its support for concurrency among other things

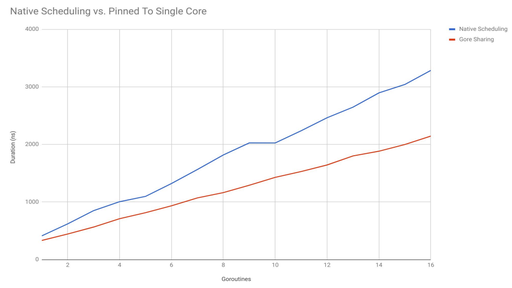
Motivation

Go has its own runtime (naive by the creators admittance) that gives provides a target for optimization



Problem

One problem we observe is that multiple Goroutines accessing shared cache lines are mapped to different cores



Approach

- However, after benchmarking several Go programs we observe that that the channels - the concurrency primitive in Go - contributes heavily to cache loads in the modified state
- We target programs that heavily rely on channels in our solution to hopefully circumvent the overhead that would come with instrumenting the scheduler for every go routine
- We instrument the runtime to keep track of the usage of all channels spawned by all Goroutines
- We periodically check if the popularity metric in our channel usage has exceeded a threshold
- If it has exceeded, we restrict the affinity of the all Goroutines to a single core
- Otherwise we let the regular Go scheduler take control of scheduling the Go routines

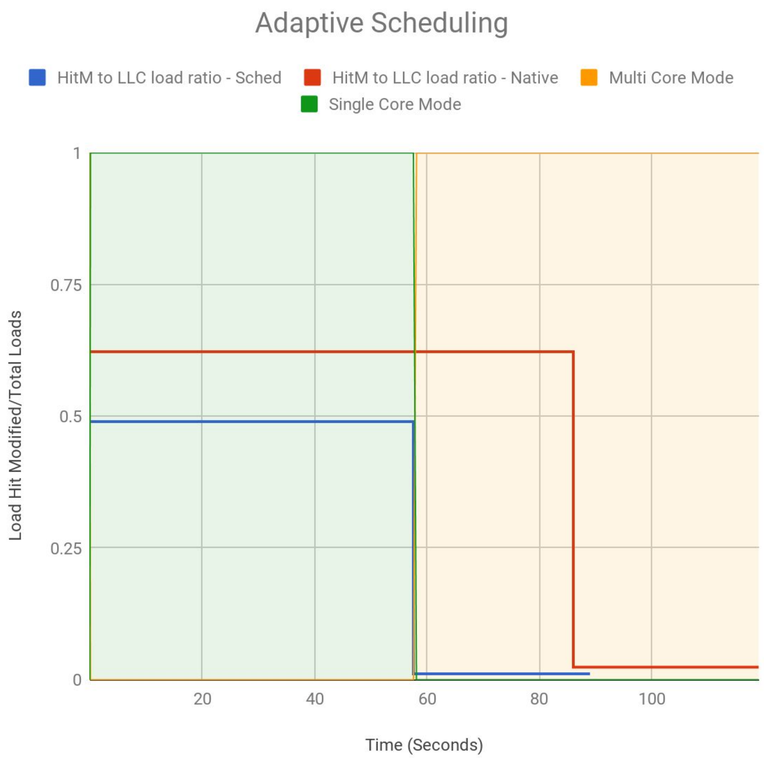
Future Work

- We observe that there is room for improvement in the scheduler which we intend to cover by:
  - Creating groups of individual Goroutines to schedule to a single core instead of the whole program
  - Instrumenting the scheduler so that we do not rely on any system calls to set core affinity
- We would also like to explore the problem of how to detect the right policy for grouping Goroutines
  - We observe that a k-means clustering type algorithm with the amount of shared accesses would be a good metric to start with, but the overhead of the algorithm may be too high
  - We intend to explore heuristics using only the hardware counter values we observe that can give a similar overall performance with hopefully less overhead

Results

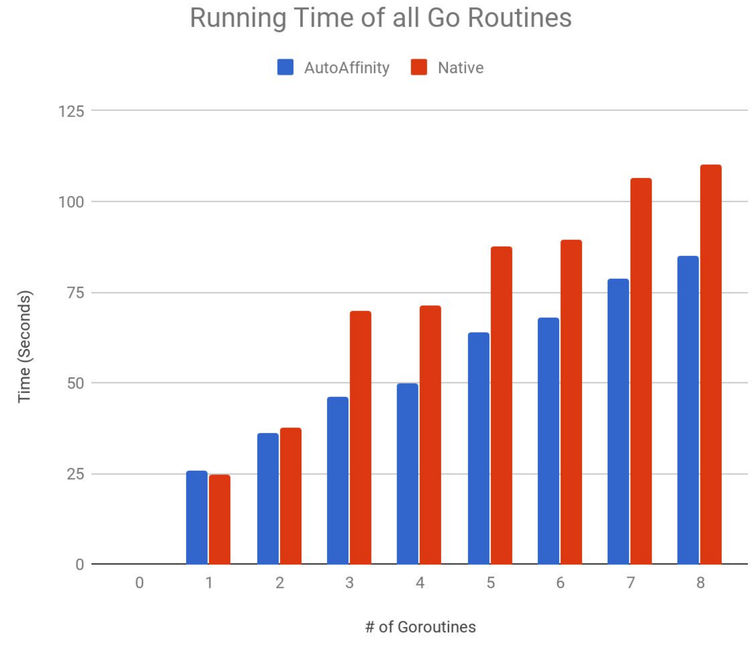
Experiment 1

- In this experiment we verify our scheduler dynamically adapts to the changing behavior of the program with regards to accesses on shared cache locations
- We spawn 8 Goroutines with communicating over 8 distinct channels
- The program has two sections
  - During the first section the program heavily uses the channel primitive for inter-Goroutine communication
  - During the second section the Goroutines do some local operations
- We compare our scheduler decisions to the cache behavior over time
- Results indicate that our scheduler component is highly reactive to the cache behavior
- We see a significant decrease in the ratio of loads that hit modified lines in the cache as well as a decrease in the overall runtime when running our scheduler



Experiment 2

- In this experiment we verify the performance due to our adaptive scheduling over the native scheduler
- We spawn a varying number of Goroutines to do a certain number of operations each and measure the total time it takes to complete
- Results indicate that the number of in the case of our adaptive scheduling, the total time is significantly slow and the difference compared to the native scheduler increases as we increase the number of Goroutines



#eliminatefalsesharing #eliminatesharing #godeep