Go Deep: Fixing Architectural Overheads of the Go Scheduler

Craig Hesling
hesling@cmu.edu

Sannan Tariq
stariq@cs.cmu.edu

May 11, 2018

1 Introduction

Golang is a programming language developed to target modern use contexts such as mult-icore programming, networked systems, massive computation clusters and web programming. In recent years, Golang has been gaining momentum, currently ranked as the 9^{th} most popular language on Github (higher than C) [1] and 19^{th} on the TIOBE index [2]. Golang is developed and heavily backed by Google, being used for parts of the Youtube and Google DL servers.

Golang abstracts the concurrency units from OS level threads to Goroutines. The crux of Go's novelty, and the main reason it can provide several advantages over other languages, lies in its Runtime component that multiplexes all Goroutines over actual OS threads. Go developers claim several advantages of this abstraction

- Context Switches do not incur expensive kernel overhead
- Goroutines are lightweight compared to OS threads in terms of memory overhead
- Goroutines have less state to setup and are thus faster in start-up times

We find that the Go Scheduler's policy of scheduling Goroutines on different OS threads, that may be running on different cores, may cause poor cache coherence interactions or false sharing overhead. Given that the scheduler is in user space and can be modified, we believe that there is an opportunity to mitigate these overheads by some hardware aware scheduling.

During our investigation, we find that the Channels construct, the main concurrency primitive that Golang provides, suffers heavily from cache coherence overhead problems. We instrument the Go Runtime to observe and change core affinity depending on the use of Channels between Goroutines. For our benchmark, we find that that our scheduler reacts relatively fast, and is able to heavily reduce cache coherence protocol overheads, resulting in shorter run times for channel using sections of a program, while not causing any significant overhead to non-concurrent sections.

2 Related Work

The Golang Runtime has been the subject of some research in recent times. [3] analyse the Golang Scheduler and suggest some points of improvement. Some of the improvements are a riteration of what Vyukov had previously suggested [4] and have now been implemented into the Runtime. A more recent design document[5] by Vyukov targets the same issue as our work: reducing the overhead of communicating Goroutines in a multi-core (or multi node) system. However, only suggestions and motivations are presented here and no ideas for overcoming the overhead this would cause are discussed. Our work tries to overcome the overhead mentioned here by targeting the channels construct in particular instead of the scheduler scehduling every Goroutine.

3 Goals

Our expected goals for this project, as given in the project proposal, were as follows

1. **75%**: Identify at least one code construct that seems to evoke a scheduling/cache problem during run-time
2. **100%**: Comprehensive analysis of the discovered problems to determine the underlying cause of these problems
3. **125%**: Suggest modifications/fixes to the scheduler to overcome the problems found

We believe that we fulfilled all three goals, and even went beyond our 125% goal by actually implementing a modification to the Golang Runtime to fix the discovered problem.

More precisely, we did the following for each of our goals:

1. **75%**: We identified the Channel construct as a source of repeated cache coherence protocol triggering by observing hardware counters.
2. **100%**: We analyzed the code implementing the Channel message passing interface and found a shared lock between all readers and writers on a channel, that would bounce around the cache if the communicating Goroutines ended up on different cores.
3. **125%**: We implemented a fix for this problem by instrumenting the runtime to adapt to channel usage by switching between single core and multi-core affinity during execution.

4 Methodology

Our Goal was to identify cases where the current Go Runtime would cause a significant overhead, due to an architectural issue, and then modify the Runtime to perform better in that case.

4.1 Channels

We target the Channels construct, which is the main concurrency primitive that Golang provides. Channels essentially provide an explicit message passing interface between multiple Goroutines.

The Channel structure contains a Queue of Senders and Receivers of in the channel. When a Goroutine has something to receive from the channel, it performs some checks on the channel structure (whether it is closed, whether it is buffered, whether there is a Goroutine in the Send queue etc.) and adds itself to the Receive linked list queue. When a sender has something to send, it performs the similar checks and adds itself to the Send linked list queue. This modification of the channel structure requires all Goroutines acting on shared channel to grab a lock, whether they are readers or writers. When a sender finds a non-empty receiver queue, the sender writes directly to the stack of the receiver Goroutine and removes it from the receiver queue. Other cases are handled in a similar manner.

4.2 Identifying Overheads

The `perf c2c` is a tool meant for analyzing programs to identify possible false sharing and cache line contention among multiple threads. We use the value of a particular hardware counter `Cache Hit Modified`(HitM), which tracks the number of hits that occurred on cache lines that were in a 'Modified' state (according to the MESI protocol). This value helps us gauge if multiple threads are causing the ownership of the cache line to bounce around among multiple cores. `perf c2c` also provides an analysis of what code is causing these particular HitM's. We used the tool on several Channel using programs and found that the channel structure lock often ended up being the offensive data value causing HitMs during execution. So we target this particular structure in our solution.

5 Instrumentation

In order to modify the Go Runtime to perform better in cases with with heavy channel usage, we do two things

1. We modify the Go Channels construct to store state to represent its usage/popularity
2. We implement functionality to monitor and change program cpu core affinity based on the channel usage

In order to store state in the Channel structure we added a new field called `touch_count`. This value is incremented each time the lock for the channel is grabbed by a Goroutine. We use this value to keep track of the channel usage.

Our periodic checker is triggered every 500 milliseconds. The checker will walk through all channel's state, determines if a given channel has high enough

activity to benefit from being placed on a single core, and then zero the touch counter. If the channel would benefit, we say that the channel votes to put the program into single core mode. If the majority of the channels vote yes for single core mode, we issue the `sched_setaffinity()` syscall to with mask value `0x1`. A channel is considered to have high activity if the touch count is greater than 800 in the 500ms window.

6 Experimental Setup

For our experiments we used two custom benchmark Go programs. The first preliminary benchmark, called `hotpotato`, was used to contrive a case where multiple Goroutines would communicate over multiple channels and then carry out an 'embarrassingly parallel' section concurrently. In the communication stage, we created a ring of Goroutines. Each Goroutine is connected to it's forward neighbor via a channel. During runtime, all Goroutines receive an integer from their backward neighbor, increment the value and pass it on to their forward neighbor. The program terminates when the value reaches a certain threshold value. The embarrassingly parallel section simply crunches a series of arithmetic operations repeatedly.

The second benchmark, which we call the `fast_regex` is supposed to mimic a real life use case of channels. It parallelizes regular expression matching over a large file by launching multiple Goroutines to handle different sections of the file. The Goroutines independently find matches in their section of the file and then report back their matches over a single channel. This is a standard message passing example, where threads only communicate when they are finished with their task.

For each benchmark, we record the running times and the internal `touch_count` values for each channels, which we are used to make our scheduling affinity decision. Additionally, we used perf to continuously record the Hardware Hit Modified Counter value (at high granularity) during the program runtime.

We carried out our experiments on an Intel Xeon E3-1505M v5 @2.80 GHz machine with 4 physical cores (8 with hyper-threading).

7 Results

7.1 `hotpotato`

We ran both the experiments for this benchmark on the 4-physical-core machine.

We ran 8 Goroutines that carried out about 5×10^7 channel communication operations over 8 channels in total and then performed some non-shared concurrent work. The results are shown in Figure 1. Note that the HITMs are practically eliminated when our auto affinity decides to trigger single core mode.

We then performed another experiment, to observe the trend over over additional concurrency, by running the same experiment with 1 to 8 Goroutines in a ring. As before, we limit the number being transfered inside the channels,

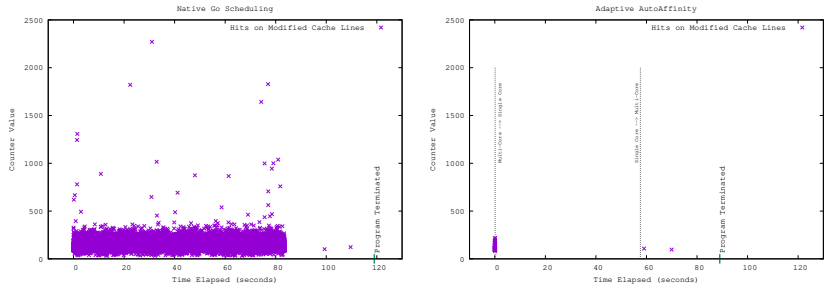

Figure 1: 'Hit Modified' Events during Preliminary Benchmark

thus all goroutines make the same number of channel transfers. compared the total running time of the case where we used our AutoAffinity Scheduler vs the Native Go Scheduler. The results are shown in Figure 2

7.2 `fast_regex`

We carried out the experiments for `fast_regex` on both the machines. We search for a reasonably complicated regular expression in a 535 MB file with approximately 10.6×10^6 lines. A simple, non-parallized implementation takes about ≈ 123 seconds to run.

7.2.1 Adaptive Scheduling

Figure 3 and 4 show results for running our `touch_count` benchmark with 8 Goroutines with our AutoAffinity scheduler turned off and on respectively. We record the Hit Modified hardware event counts and in the case of our scheduler turned on, we also track the `touch_count` value we are observing.

7.2.2 Performance Improvement

[!htb] To quantify the performance gains with varying number of Goroutines, we carried out an experiment, running the `fast_regex` benchmark with 4 to 8 Goroutines, with our AutoAffinity Scheduling turned on or off. The results are shown in figure 5

8 Discussion

We find that the `touch_count` on our channel structures is a good indicator of the actual cache coherence operation overhead. As shown in section 7.2.1, we find that the Hit Modified events, which trigger the Cache Coherence protocol, are greatly reduced by our AutoAffinity scheduler. We also observe that as soon as the Hit Modified events reduce in frequency, we change affinity back to multiple cores as in Figure 1, allowing the program to take full advantage of its

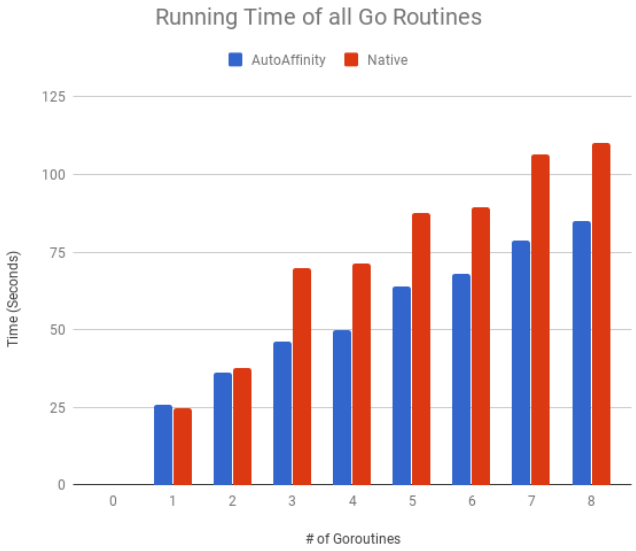

Figure 2: Completion time for `hotpotato` with increasing number of Goroutines

inherent parallelism. In the case of our `fast_regex` benchmark, we find that there are some Hit Modified events that our scheduler does not observe. We investigated and found that these were being caused by the memory allocation engine of Go. However, as soon as channel usage began, our scheduler kicked in and reduced the Hit Modifies.

An important thing to note here is that despite the extra state checking we add into the Runtime, we do not have cause a significant overhead on the 'embarrassingly parallel' section of the program, so our solution will at least not hurt performance, even if it doesn't necessarily improve it. It should also be noticed that we are able to greatly reduce the time of the channel usage section in our benchmark by setting affinity to a single core.

We can clearly see the benefit of our AutoAffinity scheduler during in Figure5. In the single Go Routine case, we might be getting some gain because there might be some Runtime Go Routines that are benefited by our AutoAffinity scheduling. We also see that we have a clear gain with increasing number of Go Routines, as the more and more Go Routines contend on a channel, we get more benefit of putting them on a single core.

9 Conclusion & Future Work

We set out to identify points of optimization in the Go Runtime with architectural knowledge. We found that the Golang concurrency primitive was a hotspot for cache overheads due to heavy mutex sharing among Goroutines us-

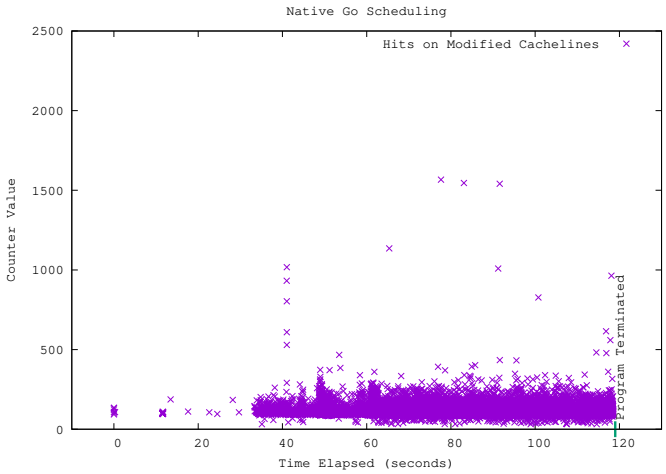

Figure 3: Native Scheduler Hit Modified events and `touch_count` on 4-physical-core machine

ing specific hardware counters. We determined the amount of times a channel is used as a reasonable alias for how much cache coherence overhead it is likely to cause and implemented a system to monitor this value and change the core affinity of the program depending on the value. Our evaluation shows that our modification is able to provide significant improvements in real world use cases.

Although we tested our solution on a dual socket machine, we would like to see the benefits of being able to set affinity to a single NUMA node instead of a single core. We expect to see some interesting results as some cache coherence problems will still exist, but these operations will be cheaper within a single NUMA node. Additionally, we would get the benefit of multiple cores within that NUMA node instead of making the program completely sequential as is the current case. This would allow programs with more diverse workloads to take advantage of our solution.

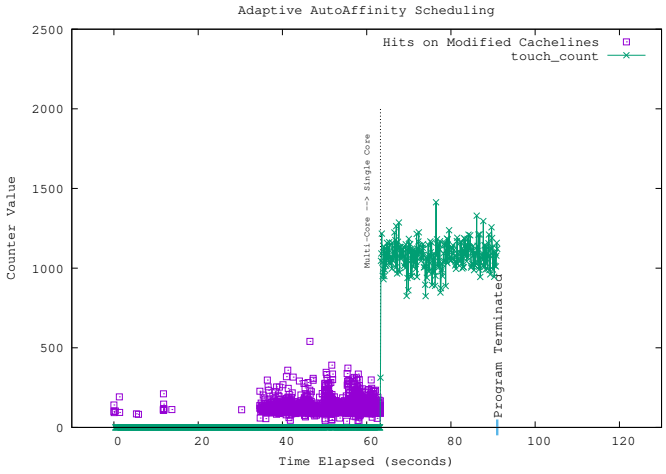

Figure 4: AutoAffinity Scheduler Hit Modified events and `touch_count` on 4-physical-core machine

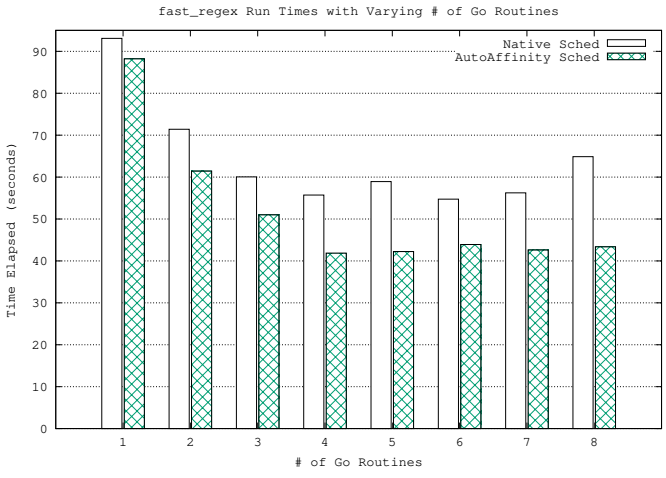

Figure 5: Run Times for `fast_regex` with varying number of Goroutines

References

[1] "Github octoverse 2018." https://octoverse.github.com. Accessed: 2018-04-10.

[2] "Tiobe index." https://www.tiobe.com/tiobe-index/. Accessed: 2018-04-10.

[3] N. Deshpande, E. Sponsler, and N. Weiss, "Analysis of the go runtime scheduler," *URl: http://www.cs.columbia.edu/˜ aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf (visited on 2016-12-19)*, 2012.

[4] "Scalable go scheduler design doc." https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit#heading=h.mmq8lm48qfcw. Accessed: 2018-03-10.

[5] "Numa aware go scheduler." https://docs.google.com/document/u/0/d/1d3iI2QWURgDIsSR6G2275vMeQ_X7w-qxM2Vp7iGwwuM/pub. Accessed: 2018-04-10.