Analysis of the Go runtime scheduler

Neil Deshpande
Columbia University
nad2135@columbia.edu

Erica Sponsler
Columbia University
es3094@columbia.edu

Nathaniel Weiss
Columbia University
ndw2114@columbia.edu

ABSTRACT

The Go runtime, as well as the most recently proposed changes to it, draw from previous work to improve scalability and performance. In this paper we explore several examples of previous research, some that have actively influenced the Go runtime, and others that are based on similar guiding principles. We propose additional extensions to the runtime based on contention aware scheduling techniques. We also discuss how such changes would not only leverage the proposed improvements currently in the works, but how they can potentially improve the effectiveness of the runtime's scheduling algorithm.

1. INTRODUCTION

The model of computation used by the Go language is based upon the idea of communicating sequential processes put forth by C.A.R. Hoare in his seminal paper published in 1978 [10]. Go is a high level language with many of the constructs proposed in Hoare's paper, which are not found in the C family of languages, and are easier to reason about than locks and semaphores protecting shared memory. Go provides support for concurrency through goroutines, which are extremely lightweight in comparison to threads, but can also execute independently. These goroutines communicate through a construct known as channels, which are essentially synchronized message queues. The use of channels for communication, as well as first class support for closures, are powerful tools that can be utilized to solve complex problems in a straightforward manner.

Go is a relatively young language, and its first stable version was released recently[8]. It is still under development, and many improvements are still being made to the language, especially to its compilers and infrastructure. In addition to the contributions of Hoare and the languages that have preceded Go, there is a wealth of other information and research that could be beneficial if applied to the Go runtime. During the course of our research we have come across many papers that share similarities with the implementation of Go, as well as some papers detailing algorithms and solutions that could easily be applied to Go. Based on this research, we have formulated an extension to the Go runtime that we believe could improve the implementation that has been proposed by Dmitry Vyukov [13].

In this paper, we are mainly concerned with exploring Go's runtime scheduler. We are interested in Go's runtime in part because we believe that relatively simple modifications to this module can result in significant performance gains. The contributions of this paper are an explanation and analysis of the Go runtime scheduler, a brief overview of existing research that relates to the Go runtime scheduler, and a proposal for an extension to the scheduler.

Section 2 presents a brief history of the Go language. We then explore the implementation of the runtime scheduler in section 3, as well as some of its current limitations in section 4. The changes that have been proposed to address the scheduler's limitations are detailed in section 5. Section 6 then describes several research papers that are applicable to the Go runtime. We then discuss the persistence of good ideas in section 7 and offer a proposal for the extension of the Go runtime in section 8. The paper concludes in section 9.

2. A BRIEF HISTORY OF GO

Hoare's paper, entitled "Communicating Sequential Processes" [10] was published before multiple processors in a single machine were commonplace. Many researchers, including Hoare, saw the precursors to this trend and tackled research questions that would need to be answered before multi-core processors could become ubiquitous. Hoare saw potential problems with communication between processes executing concurrently on separate processors. The model at the time for communication included many of the same primitives for thread communication today; namely, modifying shared memory with the assistance of locking mechanisms to protect critical regions. This model is difficult to reason about, and therefore, is prone to bugs and errors. Hoare's proposed solution included a separate set of primitives to foster message passing between processes, instead of altering shared memory.

Many of the primitives used in Go can find their origin in Hoare's CSP paper. For example, the use of Goroutines, channel communication, and even the select statement were described by Hoare (although referred to by different names). The CSP paper details many common computer science and logic problems, as well as their solutions using communicating processes. Some of the problems explored in the paper include computing factorials, the bounded buffer problem, dining philosophers, and matrix multiplication. Although Hoare's notation is vastly different, the implementation of the solutions is very much the same as it would be in Go. At the time, Hoare's proposal of CSP primitives was purely theoretical, but now that technology has advanced,

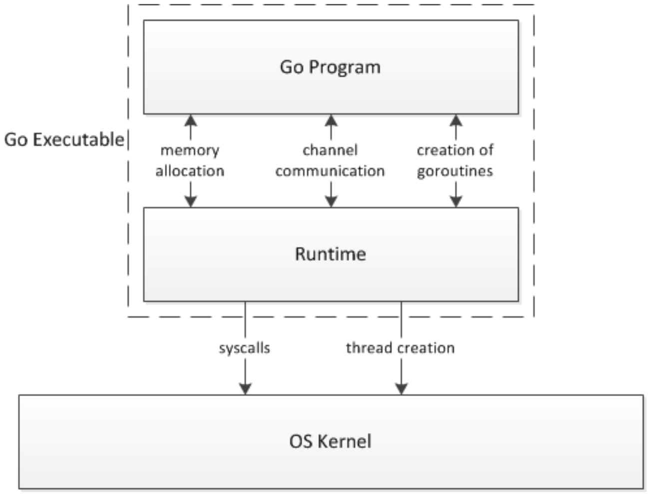

Figure 1: Diagram of the relationships between the runtime, OS, and programmer defined code

we can see that his ideas for concurrent processing were valuable and continue to be relevant almost 35 years later.

Newsqueak,a prominent member in the long line-up of CSP based languages developed at Bell Labs [4], had an important influence on Go. Rob Pike worked on several of these languages, and Newsqueak was the first in that family (Pan, Promela, Squeak) to have first class channels. This enabled the elegant composition of channels and functions to develop more complex communication structures. The study of the Newsqueak and its derivatives, such as Alef and Limbo, provides a fascinating view of language evolution, and one can trace the lineage of many of Go's elegant constructs.

3. DISCUSSION OF THE GO RUNTIME

The Go Runtime manages scheduling, garbage collection, and the runtime environment for goroutines among other things. We will focus mainly on the scheduler, but in order to do that, a basic understanding of the runtime is needed. First we will discuss what the runtime is, especially in the context of how it relates to the underlying operating system and the Go code written by the programmer.

Go programs are compiled into machine code by the Go compiler infrastructure. Since Go provides high level constructs such as goroutines, channels and garbage collection, a runtime infrastructure is required to support these features. This runtime is C code that is statically linked to the compiled user code during the linking phase. Thus, a Go program appears as a standalone executable in the user space to the operating system. However, for the purpose of this paper, we can think of a Go program in execution as comprised of two discrete layers: the user code and the runtime, which interface through function calls to manage goroutines, channels and other high level constructs. Any calls the user code makes to the operating system's APIs are intercepted by the runtime layer to facilitate scheduling, as well as garbage collection [9]. Figure 1 shows the relationship between a Go program, the Go runtime, and the underlying operating system.

```
struct  G
{
  byte*   stackguard; // stack guard information
  byte*   stackbase;  // base of stack
  byte*   stack0;     // current stack pointer
  byte*   entry;      // initial function
  void*   param;      // passed parameter on wakeup
  int16   status;     // status
  int32   goid;       // unique id
  M*      lockedm;    // used for locking M's and G's
  ...
};
```

Figure 2: Relevant fields of the G struct

Arguably, one of the more important aspects of the Go runtime is the goroutine scheduler. The runtime keeps track of each goroutine, and will schedule them to run in turn on a pool of threads belonging to the process. Goroutines are separate from threads but rely upon them to run, and scheduling goroutines onto threads effectively is crucial for the efficient performance of Go programs. The idea behind goroutines is that they are capable of running concurrently, like threads, but are also extremely lightweight in comparison. So, while there might be multiple threads created for a process running a Go program, the ratio of goroutines to threads should be much higher than 1-to-1. Multiple threads are often necessary to ensure that goroutines are not unnecessarily blocked. When one goroutine makes a blocking call, the thread running it must block. Therefore, at least one more thread should be created by the runtime to continue the execution of other goroutines that are not in blocking calls. Multiple threads are allowed to run in parallel up to a programmer defined maximum, which is stored in the variable GOMAXPROCS[6].

It is important to keep in mind that all the OS sees is a single user level process requesting and running multiple threads. The concept of scheduling goroutines onto these threads is merely a construct in the virtual environment of the runtime. When we refer to the Go runtime and scheduler in this paper we are referring to these higher level entities, which are completely separate from the operating system.

In the Go runtime, there are three main C-structs that help keep track of everything and support the runtime and scheduler:

THE G STRUCT
A G struct represents a single goroutine[9]. It contains the fields necessary to keep track of its stack and current status. It also contains references to the code that it is responsible for running. See figure 2.

THE M STRUCT
The M struct is the Go runtime's representation of an OS thread[9]. It has pointers to fields such as the global queue of G's, the G that it is currently running, its own cache, and a handle to the scheduler. See figure 3.

THE SCHED STRUCT
The Sched struct is a single, global struct[9] that keeps track of the different queues of G's and M's and some other infor-

```
struct M
{
  G*        curg;              // current running goroutine
  int32     id;                // unique id
  int32     locks;             // locks held by this M
  MCache    *mcache;           // cache for this thread
  G*        lockedg;           // used for locking M's and G's
  uintptr   createstack [32];  // Stack that created this thread
  M*        nextwaitm;         // next M waiting for lock
   ...
};
```

Figure 3: Relevant fields of the M struct

```
struct Sched {
  Lock;              // global sched lock.
                     // must be held to edit G or M queues

  G *gfree;          // available g's (status == Gdead)
  G *ghead;          // g's waiting to run queue
  G *gtail;          // tail of g's waiting to run queue
  int32 gwait;       // number of g's waiting to run
  int32 gcount;      // number of g's that are alive
  int32 grunning;    // number of g's running on cpu
                     // or in syscall

  M *mhead;          // m's waiting for work
  int32 mwait;       // number of m's waiting for work
  int32 mcount;      // number of m's that have been created
   ...
};
```

Figure 4: Relevant fields of the Sched struct

mation the scheduler needs in order to run, such as the global Sched lock. There are two queues containing G structs, one is the runnable queue where M's can find work, and the other is a free list of G's. There is only one queue pertaining to M's that the scheduler maintains; the M's in this queue are idle and waiting for work. In order to modify these queues, the global Sched lock must be held. See figure 4.

The runtime starts out with several G's. One is in charge of garbage collection, another is in charge of scheduling, and one represents the user's Go code. Initially, one M is created to kick off the runtime. As the program progresses, more G's may be created by the user's Go program, and more M's may become necessary to run all the G's. As this happens, the runtime may provision additional threads up to GOMAXPROCS. Hence at any given time, there are at most GOMAXPROCS active M's.

Since M's represent threads, an M is required to run a goroutine. An M without a currently associated G will pick up a G from the global runnable queue and run the Go code belonging to that G. If the Go code requires the M to block, for instance by invoking a system call, then another M will be woken up from the global queue of idle M's. This is done to ensure that goroutines, still capable of running, are not blocked from running by the lack of an available M.

System calls force the calling thread to trap to the kernel, causing it to block for the duration of the system call execution. If the code associated with a G makes a blocking system call, the M running it will be unable to run it or any other G until the system call returns. M's do not exhibit the same blocking behavior for channel communication, even though goroutines block on channel communication. The operating system does not know about channel communication, and the intricacies of channels are handled purely by the runtime. If a goroutine makes a channel call, it may need to block, but there is no reason that the M running that G should be forced to block as well. In a case such as this, the G's status is set to waiting and the M that was previously running it continues running other G's until the channel communication is complete. At that point the G's status is set back to runnable and will be run as soon as there is an M capable of running it.

4. OPPORTUNITIES FOR IMPROVEMENT

The current runtime scheduler is relatively simplistic. The Go language itself is young, which means there has not been enough time for the implementation of the language to mature past the first release. The current scheduler gets the job done, but it's simplicity lends it to performance problems. Four major problems with the current scheduler are addressed by Dmitry Vyukov in his design document[13] containing proposed improvements to the scheduler.

One problem is the scheduler's excessive reliance on the global Sched lock. In order to modify the queues of M's and G's, or any other global Sched field for that matter, this single lock must be held. This creates some problems when dealing with larger systems, particularly "high throughput servers and parallel computational programs" [13], which causes the scheduler to not scale well.

Further problems rest with the M struct. Even when an M is not executing Go code, it is given an MCache of up to 2MB, which is often unnecessary, especially if that M is not currently executing a goroutine. If the number of idle M's becomes too large it can cause significant performance loss due to "excessive resource loss ... and poor data locality"[13]. A third problem is that syscalls are not handled cleanly, which results in excessive blocking and unblocking of the M's, further wasting CPU time. Lastly, there are currently too many instances where an M will pass a G off to another M for execution instead of running the G itself. This can lead to unnecessary overhead and additional latency.

5. VYUKOV'S PROPOSED CHANGES

Dmitry Vyokov is an employee at Google. He published a document detailing some of the failings of the current runtime scheduler, as well as outlined future improvements to Go's runtime scheduler [13]. This section contains a summary of his proposed changes.

One of Vyukov's plans is to create a layer of abstraction. He proposes to include another struct, P, to simulate processors. An M would still represent an OS thread, and a G would still portray a goroutine. There are exactly GOMAXPROCS P's, and a P would be another required resource for an M in order for that M to execute Go code.

The new P struct would steal many members of the previous M and Sched structs. For instance, the MCache is moved to the P struct, and each P would have a local queue of runnable G's instead of there being a single global queue. Establishing these local queues helps with the earlier problem of the single global Sched lock, and moving the cache from M to P reduces the issue of space being unnecessarily wasted. Whenever a new G is created, it is placed at the back of the queue of the P on which it was created, thus ensuring that the new G will eventually run. Additionally, a work stealing algorithm is implemented on top of the P's. When a P does not have any G's in its queue, it will randomly pick a victim P and steal half of the G's from the back of the victim's queue. If, while searching for a G to run, an M encounters a G that is locked to an idle M, it will wake up the idle M and hand off its associated G and P to the previously idle M.

Another problem that Vyukov addresses is that of M's continuously blocking and unblocking, which incurs a lot of overhead. Vyukov aims to reduce this overhead by employing spinning instead of blocking. He proposes two kinds of spinning [13]:

> 1. an idle M with an associated P spins looking for new G's,
> 2. an M without an associated P spins waiting for available P's.
>
> There area at most GOMAXPROCS spinning M's [at any given time]

Furthermore, any idle M's that have associated P's cannot block while there are idle M's that do not hold P's. There are three main events that can cause an M to be temporarily incapable of running Go code. These events are when a new G is spawned, an M enters a syscall, or an M transitions from idle to busy. Before becoming blocked for any of these reasons, the M must first ensure that there is at least one spinning M, unless all P's are busy. This helps to solve the problem of the continuous blocking and unblocking and also makes sure that every P is currently involved with a running G, if there are runnable G's available. Thus, the overhead involved in the syscalls is also reduced by employing spinning.

Vyukov also suggests not allocating the G and stack for a new goroutine unless they are really required. He notes that we require just six words for the creation of a goroutine that runs to completion without making function calls or allocating memory. This will significantly reduce the memory overhead for this class of goroutines. The other improvement suggested is to have better locality of G's to P's, since the P on which the G was last run will already have its MCache warmed up. Similarly, it would be beneficial to have better locality of G's to M's since that would result in better affinity between the G's and the physical processors. We must remember that P's are an abstraction created by the runtime that the OS knows nothing about, whereas M's represent kernel threads. Most modern kernels will provide for affinity between threads and physical processors. Hence, better G to M locality will give us better cache performance.

6. RELATED WORK

During the course of our research, we came across several papers that contain solutions we believe could be useful if applied to the Go runtime. In this section we provide a brief overview of these papers, and describe how we envision those solutions could be leveraged.

row n-1	5	5	5	5	...	5
	⋮	⋮	⋮	⋮	⋮	⋮
row 1	3	3	3	3	...	
row 0	1	1	1	2	...	4
	P_0	P_1	P_2	P_3		$P_{GOMAXPROCS-1}$

Figure 5: Each P_i represents a P in the Go runtime. Each cell represents a single process, with similarly-numbered cells being processes in the same task force. When possible, processes in the same row will be scheduled to run at the same time.

6.1 Co-scheduling

Scheduling Techniques for Concurrent Systems, written by John K. Ousterhout in 1982 [11], introduces the idea of co-scheduling, or scheduling processes that communicate heavily during the same time slice. The paper discusses three different algorithms for scheduling process task forces, which are groups of processes that communicate heavily. The algorithm that can most readily be applied to the Go runtime is the paper's matrix algorithm.

The matrix algorithm arranges the processors in an array, similar to the proposed global array of P's in Go. A matrix is then created with a column corresponding to each P, as seen in figure 5. The number of rows is not specified by the paper, but we can assume that there will be sufficient rows to accommodate the algorithm. When a new process task force is created, the algorithm attempts to find the first row that will fit the entire task force such that each process is in its own cell. As seen in figure 5, task forces 1 and 2 fit into row 0. Task force 3 was too large to fit into the remaining space in row 0 and was consequently stored in row 1. Task force 4 was sufficiently small to fit into row 0 along with 1 and 2. The algorithm places processes in this matrix to better facilitate scheduling entire task forces at the same time. Assuming there are n rows, at time slice k, processes located in row $(k\%n)$ will be scheduled on the associated processors. If a processor is idle, whether because there is no process in the current task force for that processor or the currently running process has blocked, then a different process in that processor's column in scheduled to run.

If Go's P structs were used, instead of processors, this idea could work to schedule multiple goroutines that use the same channels simultaneously. This has the potential to reduce the time spent blocking M's and G's, however, this may require significant changes to the channel infrastructure.

6.2 Contention Aware Scheduling

6.2.1 Cache Conflicts and Processes

Threads executing on the same processor share cache space. Depending on the cache usage of each thread this sharing could be harmonious, or it could cause significant performance degradation. The idea presented by Blagodurov et al in [12] is to schedule threads so as to minimize cache contention on each processor. There are two main elements to such a system: the first is an algorithm to estimate the contention between threads, and the second is an algorithm to then schedule them effectively across processors.

Several algorithms were proposed in the paper to determine which threads were contentious. The most effective algorithm was one in which a profile of the cache was kept for each thread, and every cache access as well as cache miss was tracked. Picking threads to run on the same processor was accomplished by minimizing the overlap between cache usage among grouped threads. While this was the most effective solution it was also one of the most expensive. Cache miss rate was identified as a relatively effective and very efficient alternate measurement to predict cache contention between threads, and it could easily be monitored throughout the runtime of each process. The scheduling algorithm grouped threads so as to minimize the sum of cache miss rates on each processor, and assigned threads to a processor by manipulating the run queue.

6.2.2 *Contention aware multithreading for Java*

Xian et al[7] suggest an approach where the runtime scheduler proactively tries to reduce contention for locks between threads by clustering threads that compete for the same locks and then creating a linear schedule for each cluster. Each schedule can then be run on a separate processor. Their contention aware scheduler also gives higher priorities and execution quanta to threads holding locks, thus solving the problem of priority inversion.

The implementation of this contention aware scheduler is carried out by modifying the JVM as well as the Linux kernel. The JVM modifications include changes to record synchronization events, in the so called synchornization event vectors (or seVector's for short). The seVector's are then used to create the contention vectors (or conVector's), which are a measure of the contention for each shared lock. The threads are grouped into clusters based on the similarity of their conVector's, which is calculated based on a heuristic. The clustering algorithm also classifies the clusters into strong-contention or weak-contention clusters. Finally, the clusters are mapped to physical CPUs by their mapping algorithm, which attempts to balance the load across multiple CPUs by merging or splitting the weak contention clusters.

The kernel modifications include the addition of system calls to register Java threads, and to map them to CPUs. A separate contention aware scheduler which is used for scheduling Java threads is also added to the kernel. The scheduler uses a Critical Section First policy, which is an augmented priority based round robin policy, wherein thread priority is increased based on the number of locks that the thread holds, and higher priority threads get longer time quanta.

6.3 Occam π

The Occam π language is based on formalisms proposed in Milner's π calculus and the communicating sequential processes proposed by Hoare. Ritson et al[3] implemented multicore scheduling for lightweight communicating processes in the Occam language back in 2009. They utilize runtime heuristics to group communicating processes into cache affine work units, which are then distributed among physical processors using wait free work stealing algorithms.

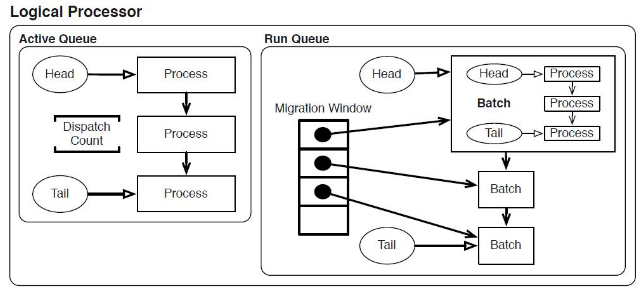

Figure 6: Schematic of a logical processor as proposed by Ritson et al. in [3]

The state of a process is stored in its process descriptor. The model that they use for scheduling is as follows: Each physical processor has a one-to-one mapping with a logical processor. As defined by Ritson et al. in [3]:

> The logical processor has a run queue, which is a linked list of batches. A batch is a linked list of process descriptors. The logical processor executes a batch by moving it to its active queue. A dispatch count is calculated based on the number of processes in a batch (multiplied by a constant) and bounded by the batch dispatch limit. The dispatch count is decremented each time a process is scheduled. When the dispatch count hits zero, any processes leftover in the active queue are moved to a new batch, which is added to the end of the run queue.

Batches are essentially groups of processes which are likely to access similar memory locations because they communicate or synchronize with each other. As discussed in section 6.2.2, threads (analogous to processes in this case) are considered highly contentious when they compete for the same locks. This contention manifests itself in the form of mutual exclusion or blocking on some communication primitive. We can form groups of processes which meet the condition that only one process in its group can be active at any given time. We also note that these groups are dynamic in nature and that their composition may change over time. Ritson et al postulate that if a batch can meet the condition of only one process capable of being active, it is probably optimal. Conversely, batches that do not satisfy this condition should be split, which can be implemented in constant time by putting the head process of the active queue in a new batch, and the remainder in a different one. The second claim made by Ritson et al is that repeated execution and split cycles will reduce large, unrelated batches into small, related batches.

The final feature that we will discuss about this paper is process migration: A process which blocks on communication or synchronization on one logical processor, A, can be woken up by a process running on a different logical processor, B. Unless prohibited by affinity settings, the woken up process continues execution on processor B. A logical processor which runs out of batches to execute may steal batches from other logical processors. However, the run queue is private to each logical processor. Hence, to allow work stealing, a fixed size migration window allows visibility and access to the end of each run queue. The fixed size of the window allows the system to leverage wait free algorithms that provide freedom from starvation and bounded wait times, improving scalability over locks.

7. PERSISTENCE OF GOOD IDEAS

Go can trace many of it's core concepts back to ideas presented in Hoare's CSP paper[10], proving that a really good idea can stand the test of time. In addition to the direct lineage, aspects of additional research can be seen reflected in Go. Portions of the Emerald language[1] resurfaced in Go, though the creators of Go were not familiar with Emerald at the time[5]. It appears, in this case, that two separate groups of researchers happened to come up with the same great idea in isolation from each other. Though, given that Emerald had been around for quite some time prior to the creation of Go, it is possible that the ideas had an indirect influence on Go's creators. Either way, the fact that the same idea appeared in different languages separated by decades, and on extremely different technology bases, shows just how powerful a really good idea can be.

Developed in the early 1980's, Emerald lists among its goals as they relate to types: "A type system that was used for classification by behavior rather than implementation, or naming."[1]

Consequently, Emerald supported both parametric as well as inclusion polymorphism, as defined by Cardelli and Wegner in [2]:

> **inclusion polymorphism** - An object can be viewed as belonging to many different types that need not be disjoint.
> **parametric polymorphism** - A function has an implicit or explicit type parameter which determines the type of the argument for each application of the function.

This is similar to the concept of a type implementing an interface in Go merely by defining the methods included in the interface rather than declaring this a priori as in Java or C++. The resemblance between the implementations of this concept in the two languages is uncanny.

We have already explored the influence that CSP has had, not only on Go, but on the entire lineage of similar programming languages developed by Rob Pike and his colleagues at Bell Labs[4],[8]. It bears repeating that the use of Hoare's CSP primitives, in essentially unchanged form several decades after the ideas were initially presented, is a real testament to their strength and continued applicability. Even more astounding is that these ideas were originally presented when the hardware to support such processes was still largely theoretical.

The Go language has been greatly influenced by previous work in Computer Science. Hence, we believe that when looking to improve parts of the language, such as the runtime scheduler, many great ideas can still be found by examining past research and applying those techniques to current work. Our additions to the changes proposed by Dmitry Vyukov center around this thought.

8. OUR PROPOSAL

Dmitry Vyukov's proposed updates[13] to the Go runtime stand to introduce significant performance improvements to the current scheduler. We think that there is still room for improvement, though, specifically with regards to reducing contention between G's. In our initial assessment of the runtime we identified several possible improvements, however, upon discovering Dmitry Vyokov's design document we realized that many of our ideas, as well as some additional improvements, were already being implemented. Upon reviewing the proposed changes as well as doing some additional research we determined that we can apply several techniques that are found in the literature to introduce additional performance gains. These include contention aware scheduling as discussed in section 6.2 in conjunction with the approach implemented by the Occam π runtime as described in section 6.3.

The proposed changes to the Go runtime actually set the stage quite well for the inclusion of contention aware scheduling. The concept of processors, or P's, allows us to design an algorithm for intelligently grouping G's to run on specific P's. We decided that the contention aware scheduling algorithm that takes locks into consideration, as described in section 6.2.2, is a better model for us to emulate than trying to implement a solution similar to the one discussed in section 6.2.1. Although the P structs (previously M structs) contain a cache, there is no straightforward way of measuring cache contention, and the issue is even further complicated by the new work-stealing algorithm, which could potentially introduce additional overhead to our analysis. This may be an area for future research, as it will be significantly more viable once the proposed changes are implemented and experimentation can be conducted to determine an effective cache contention measurement.

We can leverage an alternate contention aware scheduling technique by taking synchronization into account. Channels in Go work similarly to locks in other languages, in that they facilitate communication between concurrently executing code while also providing a means of synchronization. Reading from or writing to a channel is usually a blocking operation, since channels in Go are not buffered by default. Goroutines that communicate with each other on the same channel may not be able to run in parallel without excessive blocking due to, hence we can run such groups of goroutines serially on the same processor to reduce the overhead of blocking. This may also improve cache locality as related goroutines may be more likely to access the same memory locations. Therefore, we believe that the techniques proposed by Xian et al[7] are applicable to the Go runtime.

Goroutines have to call into the runtime to interact with channels, in part because depending on the state of the channel these interactions could have scheduling implications. Therefore, it would be relatively straightforward to record these requests from goroutines and maintain a mapping in the runtime between goroutines and the channels they have used for communication. Goroutines which communicate on

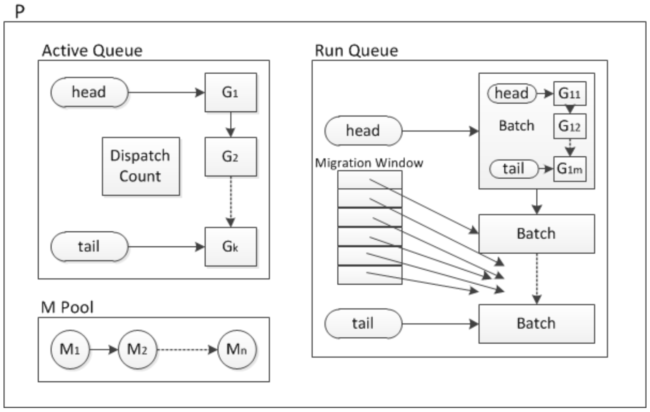

Figure 7: Schematic of our proposed structure for P's

the same channels should then be grouped together and run on the same processor. If necessary, the groups could evolve over time based on changes to their communication patterns. Once groups are established, contention aware scheduling could be integrated with the work-stealing algorithm. When a P picks a processor to steal G's from, it would need to ensure that it is stealing whole groups of related G's, much like the batches of related processes as discussed in section 6.3.

The modifications to the runtime that have been suggested above are extremely similar to the implementation of the Occam π[3] scheduler as discussed in section 6.3. We deal with essentially the same concerns, namely, trying to group related goroutines into clusters, mapping clusters to physical processors and scheduling each cluster serially on its assigned processor.

A big difference between the Occam π runtime and a solution for the Go scheduler is the mapping of P's to physical processors. In Occam π the mapping of physical processors to logical processors is known, and definite. However, as was discussed in section 5, that is not the case in Go. Therefore, maintaining high G to P affinity is not completely sufficient as we have no way to ensure that the G's are actually running on the processors that we intended. Our plan to mitigate this is to ensure a high M to P affinity by attaching a pool of M's to each P. Furthermore, we need to pin the kernel level threads that these M's represent to the respective physical processors that are represented by the P's. For each processor up to GOMAXPROCS we can create an M pool where each M in the pool is bound to the same processor. Scheduling a group of G's across this pool of M's will ensure that the G's are running on the same physical processor. A key difference between this implementation and Vyukov's proposal[13] is that M's can no longer be attached to different P's.

Due to the introduction of M pools, each M must now ensure that there is at least one spinning M in its pool, before it can block. If this condition is not met, it must spawn a new M and add it to the pool. If an M is spinning and it observes that all batches of G's attached to its P are blocked, then it must notify its P to initiate the work stealing algorithm.

As discussed earlier, our proposed changes are based on the implementation of the Occam π[3] runtime, and the similarity can be seen in figure 7. An important difference is the addition of the pool of M's assigned to a P. Since we plan to leverage the wait free algorithms proposed by Ritson et al[3]. for drawing related G's into batches and for work stealing, we have modified the structure of a P to conform to that of the logical processor in section 6.3.

There are several obvious gains to this approach. The main benefit is that we now have good affinity between G's and P's, and between M's and P's. The pinning of the M's to the physical processors ensures that we get good cache performance, since we now have a good affinity between G's and physical processors. The downside of this approach is that we are now much more susceptible to the operating system scheduling decisions because the Go runtime is probably not the only process running, and M's are no longer portable across processors. Some processors may be heavily loaded with computationally intensive tasks (external to the Go program), which will cause the M's pinned to those processors to run much slower. However, if some processors are heavily loaded, the lightly loaded processors will steal groups of G's and should mitigate the adverse effects of unfavorable scheduling decisions made by the operating system.

One scenario in which the work stealing algorithm may not be able to correct processor imbalance is when GOMAXPROCS is less than the number of physical processors, and we are unlucky enough that the M's the runtime has provisioned are pinned to the more heavily loaded processors. Another edge case in which our scheduling algorithm will perform poorly is when we are given many G's that are locked to certain M's by the user. This will cause both the clustering and the work stealing algorithms to break down. One way of mitigating this may be to switch to the old scheduler when the number of locked goroutines exceeds a given threshold which can be determined experimentally.

In summary, our major addition to enhance the scheduler is the inclusion of contention aware scheduling. This is accomplished by leveraging the batching algorithm from the Occam π implementation[3], and the pinning of M's to physical processors. We achieve the grouping of G's into contention aware batches by tracking communications through channels, and then schedule G's that communicate with each other serially on the same processor. Threads are pinned to processors, and the M's representing these threads are grouped into processor-specific M pools. This solidifies the association between P's and physical processors. By scheduling G's accross the M pools, we increase the affinity of G's to physical processors. Our model for many of these improvements is Occam π, as the implementations are extremely similar.

9. CONCLUSION

In this paper we explored the Go runtime, specifically focusing on the scheduling module. We presented a brief history and overview of Go's scheduler, and highlighted some potential areas for improvement. Some of these areas have been addressed by Dmitry Vyukov, who is in the process of updating the runtime. Based on these proposed changes, as well as a brief examination of several systems that relate to the Go scheduler, we have identified parts of his proposal which we believe can be further improved. These improvements require relatively minor changes to the runtime, and we expect that they will result in significant performance gains. In the last section we outlined our suggestions for extending the Go scheduler, and analyzed some of the implications this would have on the current system including its pending improvements.

Go's model of computation is very powerful, and much of this power comes from the implementation of the runtime, including the scheduling module. Improvements in this area can further the development of Go, thereby increasing its staying power. It takes time for a language to evolve, and even the languages that have been popular for decades are still improving. We hope that the ideas expressed in this paper will be a small step in helping Go become a popular, *Go*-to language.

10. REFERENCES

[1] Eric Jul Andrew P. Black, Norman C. Hutchinson and Henry M. Levy. The development of the emerald programming language. In *SIGPLAN conference on History of programming languages*, pages 11–1–11–51. ACM, June 2007.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[3] Adam T. Sampson Carl G. Ritson and Frederick R. M. Barnes. Multicore scheduling for lightweight communicating processes. *Lecture Notes in Computer Science*, 5521:163–183, 2009.

[4] Russ Cox. Bell labs and csp threads. http://swtch.com/ rsc/thread/.

[5] Russ Cox. Go data structures: Interfaces. http://research.swtch.com/interfaces.

[6] Dmitry Vyukov et al. Scalable go scheduler design doc discussion forum. https://groups.google.com/forum/#!msg/golang-dev/_H9nXe7jG2U/QEjSCEVB3SMJ.

[7] Witawas Srisa-an Feng Xian and Hong Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs. In *SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 163–180. ACM, 2008.

[8] Google. Go documentation. http://www.golang.org.

[9] Google. Go source code. http://code.google.com/p/downloads/list.

[10] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[11] John Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30. IEEE, 1982.

[12] Sergey Zhuravlev Sergey Blagodurov and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 28(4), December 2010.

[13] Dmitry Vyukov. Scalable go scheduler design doc. https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit.