blami: Externalized Parallelism for a Serial Web Server

Koos Kleven and Daniel Otero
Department of Computer Science and Engineering
CSE551: Operating Systems
University of Washington
{ koos42, oterod } @cs.washington.edu

ABSTRACT

Parallelizing complex services can be a daunting task for any developer. Some services are inherently limited to serial execution as a result of technological restrictions.

We've designed, implemented, and evaluated *blami*, a framework-agnostic, transparent, and entirely external parallelizing proxy and load balancer. Results show it to confer dramatic increases in performance for high-stress, parallelizable workloads. Moreover, it's flexibility, extensibility, and ease of deployment make it an ideal performance solution for applications which cannot be easily parallelized themselves.

INTRODUCTION

Ruby's rise in popularity, specifically when coupled with the Rails web application framework, has been prolific in the last few years. As of version 1.8.6, however, Ruby was an entirely interpreted language with an entirely user-level threading library. In addition to the multiprogramming short-comings of Ruby itself, the Rails framework is not remotely thread safe. This com-bination of factors makes it very difficult to leverage the ease of development provided by Ruby on Rails when making large-scale internet applications.

Mongrel, a web server made popular largely by its simplicity, is often used to serve Rails applications. Written in Ruby, Mongrel's performance suffers as a result of the aforementioned issues. First, because a threaded Ruby application ultimately sits on top of a single kernel thread, any Ruby thread that causes kernel-level blocking also brings the application's remaining threads to a grinding halt. Second, the lack of thread safety in the Rails framework led the creator of Mongrel to enforce mutual exclusion for the entirety of Rails code. While Mongrel itself is multi-threaded, no two threads can be executing Rails code at the same time, effectively killing real parallelism.

We have created *blami* (*B*alancing *L*oad *A*cross *M*ongrel *I*nstances), transparent middlebox software capable of proxying requests to and from replicated Mongrel servers for the purpose of introducing true parallelism, though parallelism external to the underlying application.

DESIGN GOALS

We maintained three core design goals during the creation of *blami*:

1. Simplicity
2. Maximum protocol-agnosticism
3. Transparency and framework externality

Simplicity

It would have been possible to optimize *blami* heavily to leverage the conventions used by Rails, or to fully take advantage of the many performance features built into the HyperText Transfer Protocol (HTTP). To do this, however, we would largely duplicate the functionality already built into the requesting browsers and responding HTTP servers.

Worse still, *blami* would become outdated anytime an aspect of Rails changed, or a feature was added to the HTTP spec.

Our system was therefore designed to be the simplest possible layer that could be inserted between current client-server HTTP endpoints. Though we sacrifice possible optimizations, and therefore peak performance, in lieu of simplicity, we dramatically ease the use, maintenance, and configuration of our system.

Protocol-agnosticism

While *blami* was intended for use with Rails, and therefore HTTP, it follows from designing for simplicity that one would also maximize the system's level of protocol agnosticism. If it were possible to open *blami* to other protocols and applications without negatively impacting the design process, it would only increase the value of such a system.

To make *blami* Rails-compatible, at the very least, put limits on how protocol-agnostic it could be. We settled for supporting any stateless, TCP-based protocol. A further, more subtle, restriction is that the protocol cannot rely on anything but the content of its messages (i.e. TCP/routing state cannot be used).

Transparency and framework externality

Due to the vast and steadily-increasing popularity of both Ruby and the Rails framework, it was unrealistic to address these problems by introducing branched changes in their core libraries. The cost of adoption would be too great, especially for clients with substantial time investmented in the existing versions. In addition, our system would impose too great an inconvenience if it somehow required that either the client or Rails application be aware of the existince of the middleware.

To avoid these issues, *blami* functions completely transparently. Neither the client nor the service are remotely aware of its man-in-the-middle behavior, nor do any changes need to be made to the client or server software or configuration.

Sacrifices

One of the major downsides to this form of process-based parallelization is that it carries much more overhead than a more purpose-built methodology. Not only does the application suffer the overhead of multiple copies of its executable in memory, but it suffers further from the duplication of work between instances of the application. For instance, if one process serves a request for resource X and caches it, that caching does nothing to increase the performance of a different process. In fact, a user could request the same resource from *blami* up to *n* times (*n* being the level of back-end replication) and, in the worst case, cause *n* different replicated processes to miss their caches.

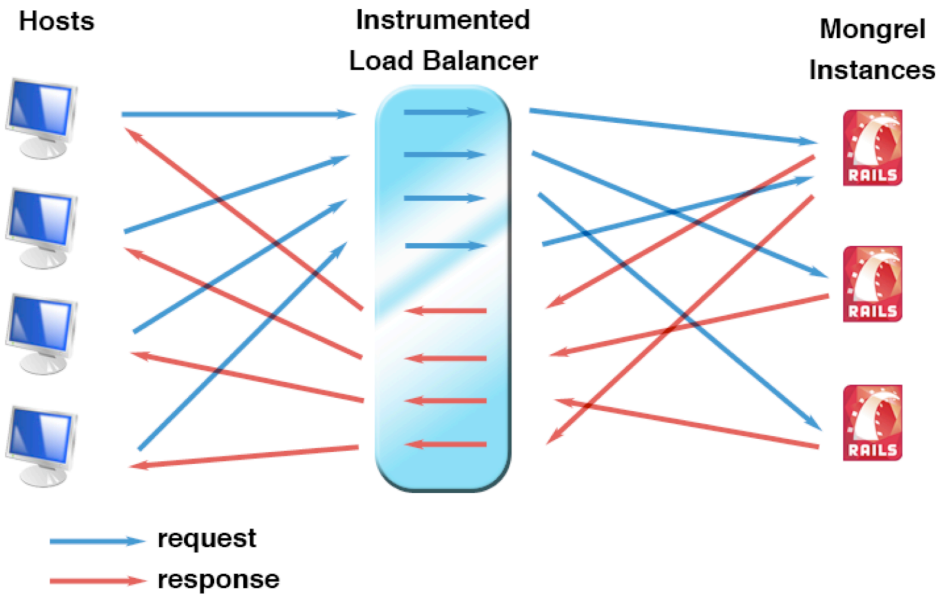
To make matters worse, *blami*'s protocol-agnosticism makes it impossible for intelligent load balancing based on *what* is being requested by a client. In the case of HTTP, the middlebox would have to understand the content of HTTP headers to a degree that it would even recognize equivalent requests, and subsequently redirect them to the same back-end server. As should be apparent, this would be fairly easy to do poorly. One could, for example, recognize resources that appeared to be static. However, to do such intelligent load balancing correctly, especially given the very dynamic nature of today's web applications, *blami* would need a tremendous amount of HTTP-specific "knowledge."

While we do require that the serviced protocol be stateless, many clients nonetheless try to be "smart." If a client application relied on being connected to the same back-end server throughout multiple requests in order to function properly, its behavior might be irregular.

System Design

Concurrency Model

Rather than using multithreading, an event-driven architecture was used instead. This was beneficial in a number of ways. First, it vastly simplified implementation, as concurrency within *blami* was not an issue. While it would be possible to run multiple proxies in separate threads, each thread would be self-contained and thus no synchronization of any kind would be required.



The event-driven architecture also has performance benefits over a threaded approach. Eliminating synchronization inherently aids performance by eliminating cycles wasted on locking and state duplicated in multiple threads. For the benefits of serial execution to be realized, however, that serial execution must not waste resources (e.g. idle CPU). By deconstructing each client-to-server round trip into its constituent non-blocking parts, and by performing other operations during blocking, our event-driven control flow efficiently multiplexes requests without substantially decreasing performance.

Architecture

blami behaves like any other web server. It accepts connections on a specified port, opens each connection on a new port, fulfills client requests over the created connections, and then closes connections when they are no longer needed.

The difference between a traditional server and *blami* is that it does none of the work to compute the appropriate response. Behind it run any specified number of replicates, the parallelized back-end servers tasked with doing the "real" computation. As soon as *blami* receives a client request, it chooses a replicate based on defined criteria (see "Load Balancing" below) and forwards the request to that replicate. When the replicate responds, *blami* then returns the response to the corresponding client.

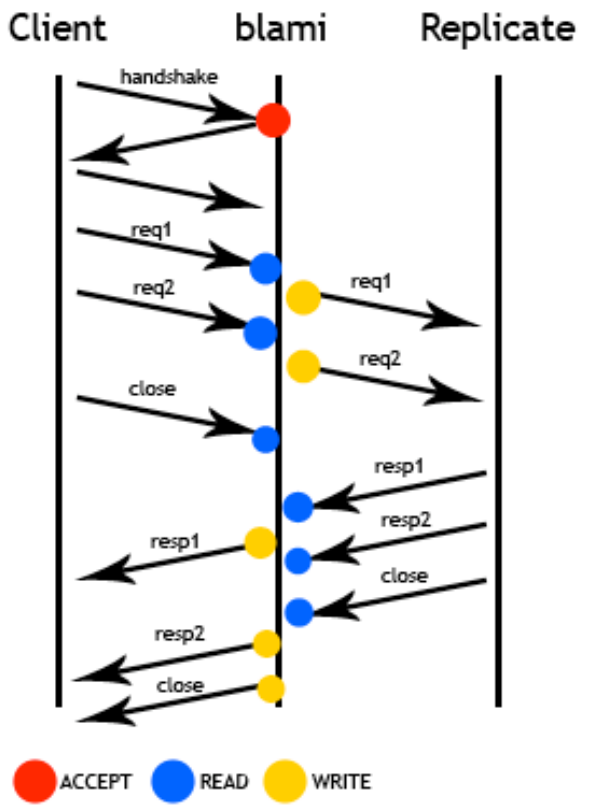
While we designed *blami* with Rails and Mongrel clusters in mind, and thus made it easy to run it on the same machine as its back-end replicates, this is by no means necessary. It would be perfectly easy to offer much greater degrees of parallization by moving replicates to different machines with their own plentiful resources. However, we anticipate that in the common case, *blami* would be used for small-to-medium deployments of Rails on a single machine.

Control Flow

There are three basic events in the *blami* event model: *accept*, *read*, and *write*. Whenever an event is triggered, it is added to a queue of events to be handled. Each event is handled serially in the order that it was generated.

When an instance of *blami*'s ZokaServer – the event-driven server that powers the system – is created, it is given a port on which to accept connections. When a connection request arrives on this port, an *accept* event is triggered. When handled, a connection is created with the requesting client, as is a connection to a replicate chosen to service this client's request. This pairing of client to replicate will exist as long as the two connections

remain open. It is then up to the client and the protocol as to when the connection should close.



When data arrives on a socket, whether its endpoint is a client or a replicate, a *read* event is triggered. Reading comes in two flavors. Either there will be data ready to be read, or what waits is an end-of-stream notification, in which case the connection must eventually be closed. When data is read, it is immediately added to the pending data queue for the corresponding connection. For instance, any data received from a replicate will be enqueued for sending back to the associated client.

If an end-of-stream request is received, the connection can be closed only *after* all relevant responses have been written back. For instance, in the diagram above, it would be inappropriate to sever the connection with the client when they request to close. If *blami* were to terminate the connection then, there would be no connection to write responses back to when they arrived from the corresponding replicate. Before severing a connection with a client, one must ensure that the corresponding replicate connection is no longer in use, and that there is no pending data waiting to be written back to the client.

Any time that a *read* occurs, with the exception of an end-of-stream message, data will have been read which needs to be forwarded to the opposing party. If reading from the client, data must be forwarded to the replicate, and vice versa. Thus, at the end of a *read*, the corresponding connection is marked to notify when it is available to write. It is when the connection becomes writable that a *write* event is triggered.

Load Balancing

In order to allow flexibility and experimentation with respect to load balancing, we built an interface for a load balancing strategy, and allow *blami* to run with any valid implementation. We were only able to create two implementations ourselves, however.

Round Robin balance strategy, not surprisingly, implements a round robin scheduling system for client requests over the available replicates. Least Loaded balance strategy, on the other hand, routes new requests to the replicate with the fewest incomplete requests pending.

EVALUATION

We carried out a series of twenty tests with different proxy settings and replication factors in an attempt to evaluate the performance of the system.

Machine Setup

Our system tests were performed using an instance of *blami* hosted on a MacBook Pro. The test machine had a dual-core Intel Core2 Duo (clockspeed 2.33Ghz) processor and 2GB of main memory. The machine ran an instance of the *blami* proxy, which in turn relied on up to ten (varied based on the test) instances of Mongrel serving the same Rails application. The Mongrel instances were also run locally on the test machine. As a further detail, the Rails application (served, again, by Mongrel) was run in development, rather than production mode. Effectively, this had the effect of turning off caching of any machine code or dynamically generated content at the replicate level. We believed that this caching was an entirely application-specific feature of Mongrel, and chose to

disallow it so as to better understand the raw performance gains offered by parallelizing. Note that Mongrel could still cache static files.

Testing Infrastructure

We used two different means to generate traffic to our proxy. At the outset of testing, we used Seattle, a distributed computation framework, to request resources as specified by particular testing workloads. Seattle granted us use of small slices of "vessels," each of which could run our test code written in a subset of Python.

Seattle proved to be problematic for our tests. The wide disparity in quality of network connections among our vessels led to dramatically unpredictable and outlier-ridden results. While our problem may have been solved by increasing the size of our vessel set, we failed to do so in a timely manner, forcing us to pursue other options.

In Seattle's stead, we made use of access to University of Washington laboratory machines spread throughout the Paul Allen Center for Computer Science and Engineering. We wrote a distributed bot using Python that would, from each of the testing lab machines, repeatedly request resources matching a given workload for a specified period of time. The bot also spaces requests by a semi-randomized interval to avoid network congestion.

Evaluation Tools

To ease data-gathering and analysis, we created Gnosis (a tongue-in-cheek suggestion of "divine insight and knowledge"). Gnosis is, itself, a Ruby on Rails application. Somewhat ironically, it is both the application to which we proxied *blami* during our tests, and the application which gathered metrics for the proxy's performance under various loads.

Connectivity to the Gnosis database is built into *blami*, and every completed request (initial client request through to full response) is logged, along with several metrics, to this database. Gnosis then has several features to allow intuitive presentation of information, and can assemble cumulative statistics over a series of requests. This system allowed us to look at throughput in both requests per unit time and bytes per unit time, average latency per request, as well as maximum, minimum, and average requests per unit time per host.

Workloads

To test the effect of our parallelism under different loads, we built several somewhat contrived workloads into Gnosis's evaluation features. These workloads included:

1. Trivial: Simply returns the text "Hi!" without going to disk or interacting with a database. We expected to see virtually no difference in performance with this workload, as it would theoretically tax the system very little.

2. Normal: Fairly light reading from a database, loading of a template from disk, and rendering of that template. We expected to see some improvement in performance with increase in parallelism, but were skeptical about the degree to which performance would improve.

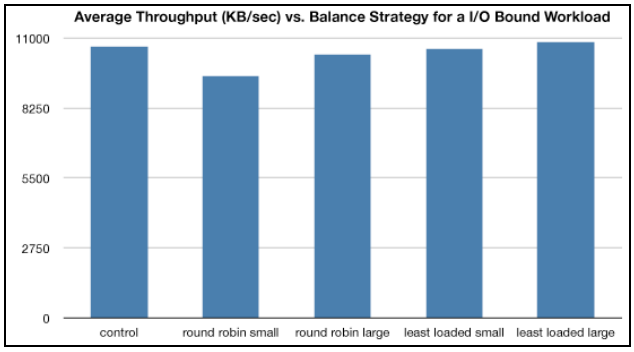
3. Disk-bound: Better referred to as IO-bound, perhaps, this workload returned a file at random from a group of one hundred 5-megabyte files. Its design was intended to make any caching of static resources impossible, forcing disk reads and long network writes. As a result, we expected very low request throughput. Because a single-threaded server would be more efficient when reading a large file from disk, we expected the serial server to perform best on this workload.

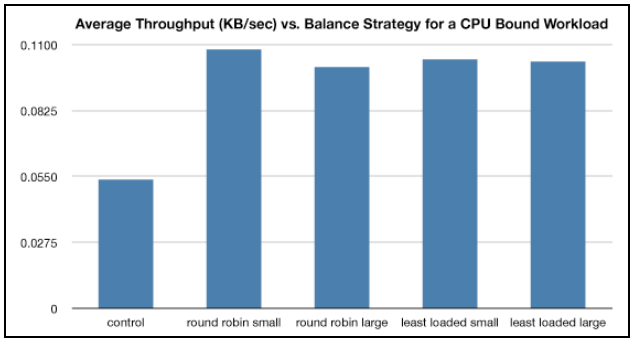
4. CPU-bound: A profoundly wasteful workload, this one simply burns processor cycles on an arduous computational task intended to take around five seconds on an unloaded machine. Parallelism, we thought, would undoubtedly improve performance on this workload.

blami Instances

We tested each one of the above workloads on five different instances of *blami*, each designed to expose the performance characteristics of a particular degree of parallelism and load balance strategy. The following table describes these instances.

Balance Strategy	Parallelism
Round Robin (control)	1 replicate
Round Robin (small)	5 replicates
Round Robin (large)	10 replicates
Least Loaded (small)	5 replicates
Least Loaded (large)	10 replicates

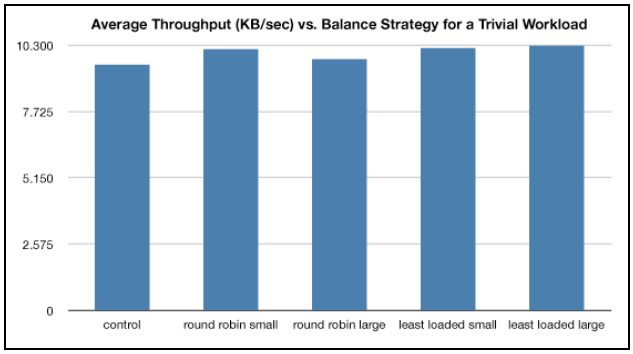


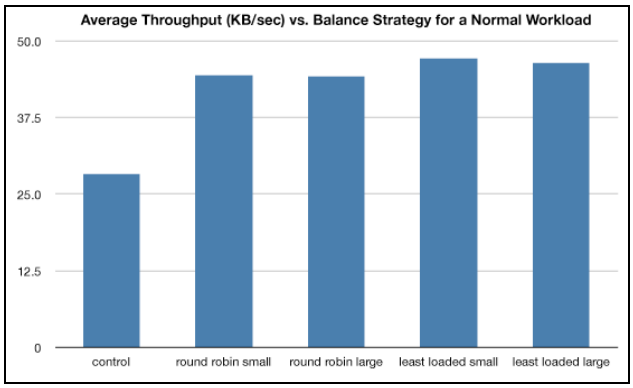


RESULTS

Performance Graphs

The following graphs summarize the observed performance of each instance over the various workloads tested. For ease of reading, larger versions are provided at the end of the paper.





Analysis

Trivial: As expected, there was little performance variation in the trivial workload. Despite servicing almost two thousands requests in less than one minute, CPU never so much as approached maximum load in even the control case. Without a sufficiently taxing load, parallelism conferred no benefits.

Normal: The big surprise in our testing was to find that the performance under our "normal" load improved dramatically when parallelism was introduced. We expected some improvement, but in the best case, request throughput improved by 71% over the control. We attribute this to the fact that the normal workload, while not nearly as taxing as the IO- and CPU-bound workloads, was sufficiently taxing so as to bring a single-threaded server to its knees. Thanks to database and other disk interaction, each request took about an order of magnitude longer than the each trivial workload request. As a result, the communication and protocol overheads per request were far lower, proportionally, allowing the benefit of parallel processes running on distinct physical processors to take effect.

Disk-bound: This was no surprise. The control outperformed the parallelized proxy instances, though not dramatically. The reasoning behind this seems fairly straightforward. When reading a large file from disk, the process goes fastest when the read is uninterrupted. Each context switch or interrupting seek not only consumes time, but also forces another disk seek to resume reading. Each seek wastes valuable time, and the disk takes a moment to ramp back to full bandwidth, making a schizophrenic concurrent proxy a poor match for this workload.

CPU-bound: While it was obvious that CPU-bound tasks would favor parallelism most, we did not see the performance we hoped. Despite twofold increases in throughput upon parallelization, there was virtually no difference in performance between instances with varying degrees of parallelization. Possible reasons for this discrepancy between hypothesized and observed behavior are discussed below.

Balance Strategies

Unfortunately, no test seemed to reveal a clear strength of one balance strategy over the other.

Pitfalls

Once we began analyzing test results it became immediately obvious that our testing machine was flawed. We saw next to no variation between degrees of parallelism, and for good reason. With five replicates (our "small" replication factor), we were already overpowering the number of physical CPUs by more than a factor of two processes per processor. When increasing to ten replicates, we weren't helping parallelism at all; we were merely further overloading our dual-core machine.

This was no more evident then in the CPU-bound workload tests. Given a ten-way parallelized application that is entirely CPU-bound, one absolutely *should* see speedup as the degree of parallelism increases, but only if the hardware supporting the application is truly parallel. Ours was severely limited.

FUTURE WORK

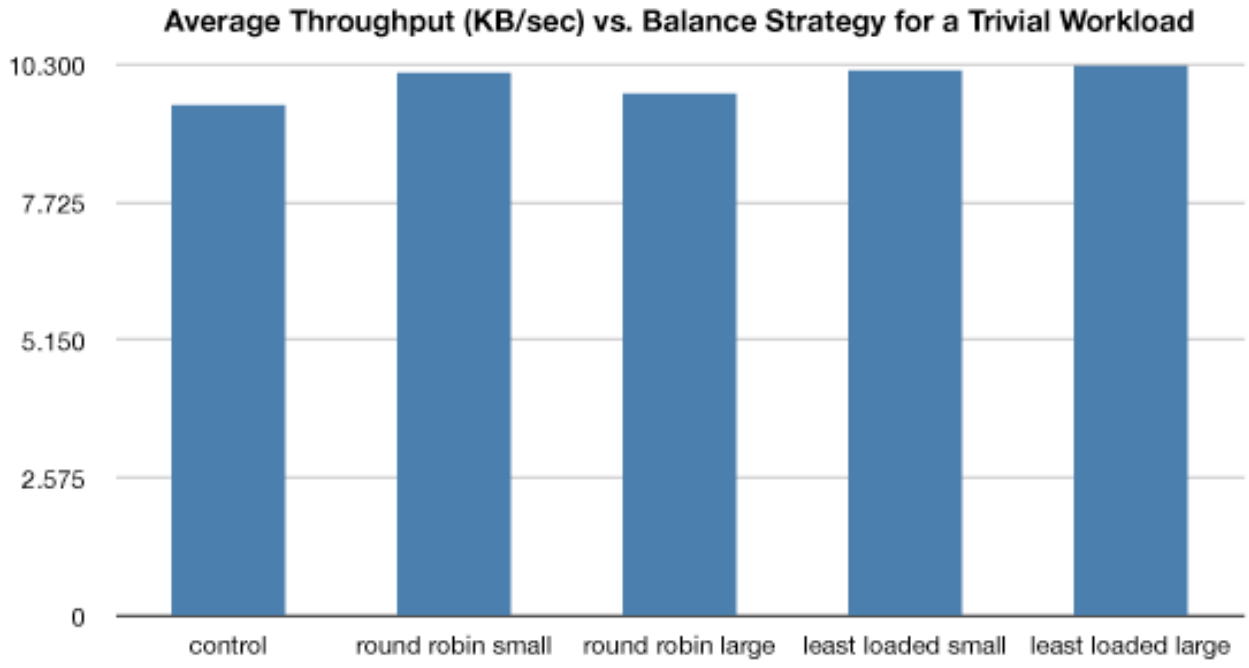
The most glaring omission in our results was that of successful non-standard load balancing strategies. We had very little luck coming up with strategies that didn't fail from the outset.

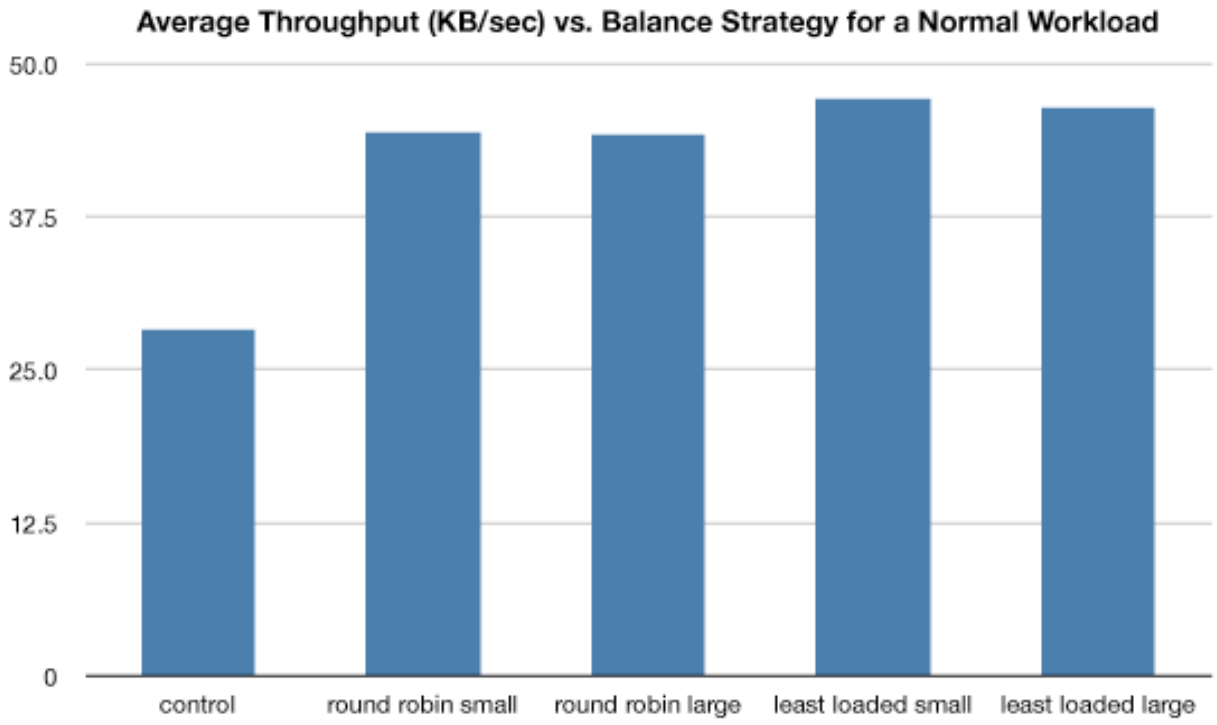
One of the obvious extensions to our existing Least Loaded balance strategy would have been to add a mechanism to enforce some form of fairness. Fairness is important in many scenarios, ranging form casual resource starvation to malicious Denial-of-Service attack. Our balance strategy interface, thanks to being able to record the start and end times of transactions, would be well-suited to building such a mechanism. An obvious approach would be to keep track of the number of pending requests per client and restrict that number to some non-threatening threshold. No doubt others exist as well.

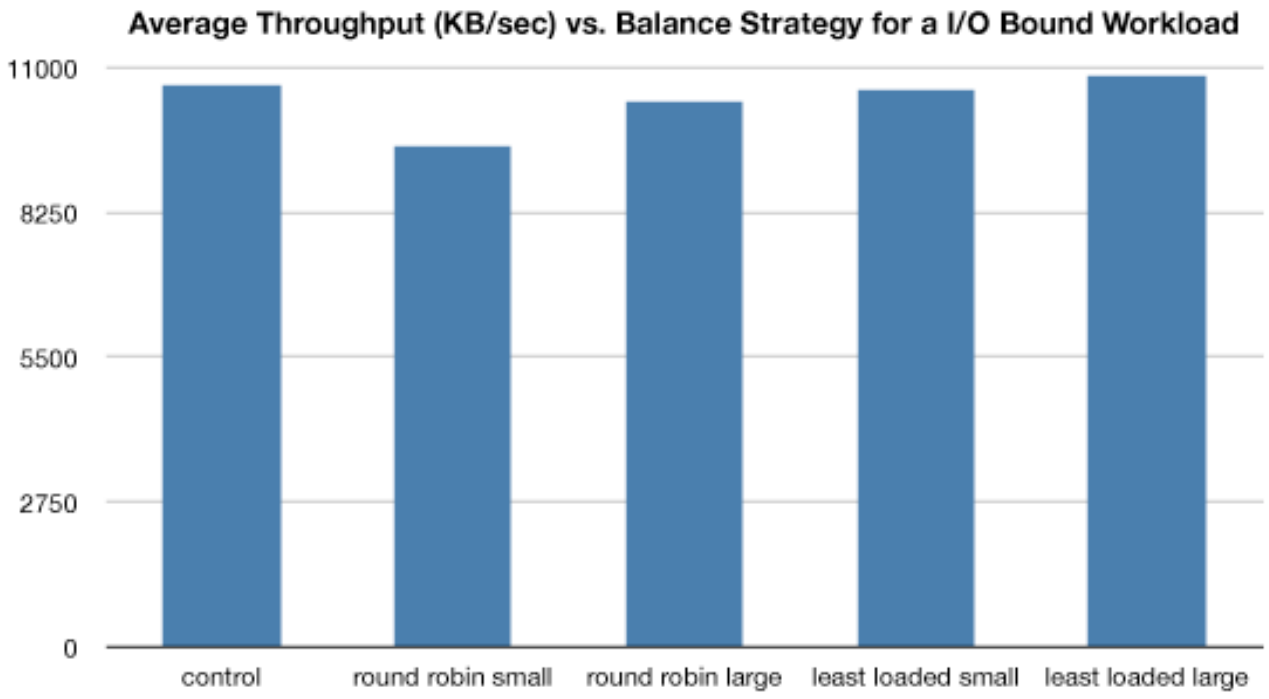
Another interesting area of research would be in breaking what limitations *blami* demands. Without ties to stateless protocols or TCP, *blami* could evolve into an entirely protocol-agnostic proxy and load balancing system. This would have all sorts of uses, including local network congestion control, mediation between different loads and protocols (e.g. bittorrent, streaming media, chat, web traffic, etc.), and exposure of multiple potentially-conflicting services through a single host address and port.

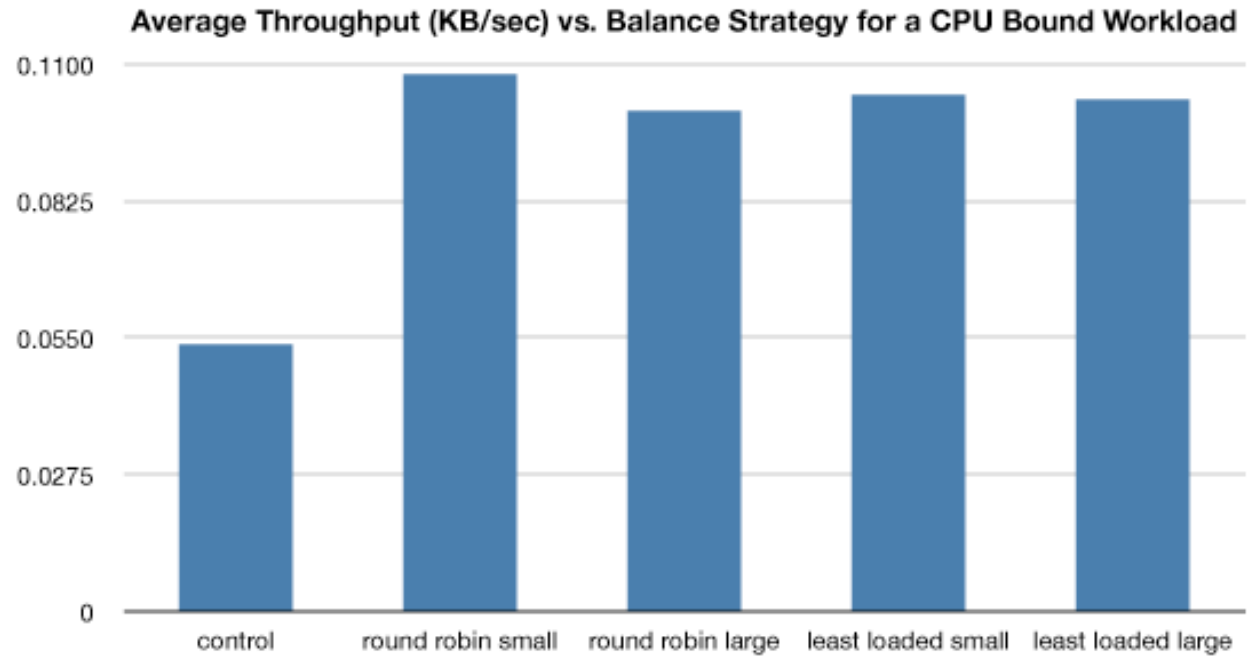
CONCLUSION

We sought to increase the performance of poorly-multiprogrammed Ruby on Rails HTTP applications by introducing parallelism over the entire application stack. We then showed that the same principle could be abstracted to serve any stateless TCP protocol. We implemented our concept in the form of *blami*, an event-driven, semi-protocol-agnostic parallelizing proxy and load balancer. Our evaluation clearly demonstrated substantial speed up over single instances of serial back-end servers, though we were unable to achieve diversity in load balancing, perhaps as a result of our limited testing equipment. Finally, we proposed opportunities for further exploration of and improvement on *blami*'s underlying approach to improving performance.


Average Throughput (KB/sec) vs. Balance Strategy for a Trivial Workload
10.300
7.725
5.150
2.575
0
control
round robin small
round robin large
least loaded small
least loaded large


Average Throughput (KB/sec) vs. Balance Strategy for a Normal Workload
50.0
37.5
25.0
12.5
0
control
round robin small
round robin large
least loaded small
least loaded large


Average Throughput (KB/sec) vs. Balance Strategy for a I/O Bound Workload
11000
8250
5500
2750
0
control
round robin small
round robin large
least loaded small
least loaded large


Average Throughput (KB/sec) vs. Balance Strategy for a CPU Bound Workload
0.1100
0.0825
0.0550
0.0275
0
control
round robin small
round robin large
least loaded small
least loaded large