Exploiting Parallelism in a Proxy Server

Vikas Natesh

ABSTRACT

This project examines opportunities for parallelism in a web proxy server using pThreads. Several techniques were chosen to make the server as efficient as possible namely thread pooling, caching of frequently used URLs, and a concurrent hash table for storing URL request to cached file mappings. After testing on several request and thread counts, the performance of the server was measured and compared against a popular open-source asynchronous-IO library, python's asyncio. The new server was able to achieve 128 requests/sec when using 64 threads, compared to asyncio's 40 requests/sec on 64 threads. It also achieved better speedup, better efficiency, and weak scalability.

INTRODUCTION

Proxy servers are used by internet users to hide their traffic from governmental authorities or ISPs. Like traditional web servers, they need to process many client requests simultaneously. However, traditional servers retrieve documents from disk and send them to the client. Proxy servers on the other hand must first perform HTTP requests on a client URL to retrieve the HTML document from the corresponding website, and then send this document back to the client. This results in heavier burden on the server.

To serve many client URL requests simultaneously, proxy servers can be parallelized. Using individual processes to handle requests is a first option. However, serving requests is an IO-bound operation and the overhead on creating/managing processes for requests is greater than the benefit gained from parallelization. Threads are lightweight and enough to serve our purposes.

While threads are simple to manage, launching a new thread per request is still wasteful. Thread creation/destruction requires expensive system calls that degrade performance. A better approach is to use a thread pool. Here, a finite number of threads are created at the outset, before any requests have arrived. New requests enter a task queue where the head of the queue checks the thread pool for any free thread not currently performing work. If a free thread is found, it is assigned to that request and marked as busy. Once the request has completed, the thread is returned to the pool and marked as free. This cycling of threads is efficient, granted that synchronization is also handled correctly. Details are described in the implementation sections.

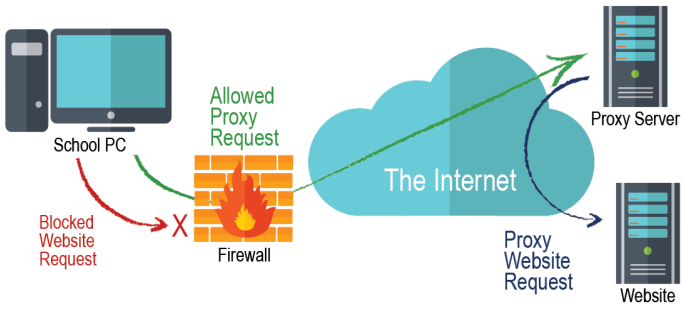


Figure 1 – Proxy Server Model

A popular optimization in web servers is to introduce a cache to store frequent requests. URLs such as 'www.google.com' are common and retrieving google web documents from disk has an order of magnitude lower latency than the mean round-trip time (RTT) required for an HTTP request [1]. In the context of multithreading, this cache must handle asynchronous access by multiple threads. Multiple threads simultaneously reading and writing to the same piece of memory can lead to race conditions and contention. Consider the case where two threads on different processors, one reading and one writing to two different slots of hash table that happen to reside on the same cache line. The first thread to succeed in writing will invalidate that cache line for the other thread, causing the coherence protocol to be applied. This situation is known as false sharing. Thus, the hash table must be aligned in such a way that different slots will end up on

different cache lines. In addition, there needs to be a collision resolution and replacement policy for stale URLs. These issues will be addressed in the implementation section.

This project contributes the following: a multithreaded proxy server using thread pooling via pThreads; an asynchronous web cache using a concurrent, cache-aware hash table; performance evaluation and comparison to Python's asyncio package; and a real-world implementation for local-area networks using socket programming.

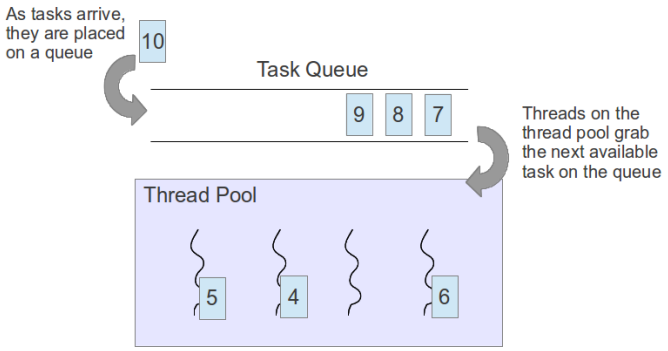


Figure 2

RELATED WORK

Several studies have been performed regarding the choice of threading library as well as cache-aware hash tables and replacement polices in proxy servers. For example, researchers tested the viability of OpenMP and found that although the request processing loop could be parallelized with few directives, it did not perform efficiently [2]. They also found that pThreads performed well, although it was cumbersome to program in as the programmer must manually deal with task distribution. Nonetheless, pThreads offers a higher degree of control due to its low-level API.

Different replacement policies in web servers have been analyzed. Popular ones include least recently used (LRU), least frequently used (LFU) and most recently used (MRU) [4, 5]. While extremely sophisticated algorithms based on deep recurrent networks have been recently proposed [3], the simple LFU algorithm works well for our purposes. The dangers of LFU, namely that the counter remains artificially high for stale URLs, are mitigated in our implementation by the fact that the entire web cache is cleared ever so often.

There is a large body of research on concurrent, cache-aware, and cache-oblivious hash tables. For instance, researchers have built hash tables that are distributed amongst CPU caches, avoid fine-grained locks, and achieve 1.6X higher throughput and less cache misses than current implementations [6]. Reducing the number of locks is thus, an important factor in building a concurrent table. Efficient collusion resolution is also important, especially when the number of unique requests greatly exceeds the table size. Linear probing has been shown to be extremely efficient as it exploits cache locality and can even be made cache-oblivious [7]. Moreover, it helps reduce false sharing scenarios described earlier. More on this strategy is discussed in the next section.

IMPLEMENTATION

This section discusses the implementation details of the proxy server that lead to several optimizations. A thread pool is used for parallelism and synchronization is controlled using pThread mutex locks. Its API is made up of several functions:

threadpool_create : This creates a pool containing a user-specified number of threads. A maximum task queue size must also be specified to prevent queue buildup.

threadpool_thread : This function creates a free thread and waits in a loop until it acquires a lock controlling access to the queue. Once it has the lock, the thread can grab a task from the queue, release the lock for other threads to use, and execute the task.

threadpool_add : This adds a new task (request URL) to the queue, provided the number of pending tasks does not exceed the maximum queue size.

Once the thread pool is created, new requests are assigned threads. Each thread executes a curl (libcurl) call on that request URL. Specifically, curl returns the result of an HTTP GET request. HTTP POST requests are not handled in simulation.

As mentioned earlier, a web cache is used to improve response time for frequent requests. This cache is represented as a hash table instantiated with zeroes. The table itself is a 1-d array where every 5 consecutive elements are interpreted as a probe sequence. Once a request has been assigned to a thread from the pool, the URL is hashed using the non-cryptographic MurmurHash3 algorithm [8]. The hash is modulo'd with a fixed table size (the array length) to yield an index. If the value found at this index is equal to the hash, the URL is already in the cache and we can retrieve it from disk. Otherwise, the hash value is inserted into the table at the index. A curl for the request is then executed and the HTML response is stored on disk using the hash value as the file name.

Asynchronous access to the table is enabled using pThread read-write locks (rw_lock). In this model, multiple threads may simultaneously read from the cache i.e. each thread can hold the lock for reading. However, if a thread successfully acquires the lock for writing, then all other threads are blocked from both reading and writing to the cache. A more fine-grained approach is to maintain a separate mutex lock for each table entry and allow asynchronous writes to the unlocked indices of the table. However, this introduces significant overhead as we must lock and unlock the mutex on every write. It also incurs storage overhead as we must now store potentially thousands of locks.

Collisions are resolved by linear probing. Upon a collision at some index, the next 5 elements are searched for an empty slot. This scheme provides the benefit of CPU cache locality of reference as consecutive elements are loaded into the same cache line. The replacement policy for the cache works in a similar fashion. Each has table entry contains not only the URL hash but also an attribute for counting the number of times a request was queried for. This is a least-frequently-used (LFU) policy and the cache is periodically purged of unused entries. In the case of an unresolvable collision when writing to the table (no empty slot found after probing 5 elements) the element with the lowest LFU counter is overwritten.

The collision and replacement rates are complex functions of the possible distributions of URLs of requests at any given time. In this report, we do not provide probabilistic guarantees or analysis on performance hits due to web-cache misses. However, these architectural decisions were empirically tested and found to be robust. The purpose of our simulations was not to test the efficacy of the replacement policies, but to test server performance under heavy load. Our characterization of heavy load is described more in the experiments section.

EXPERIMENTS

In this section, we describe several experiments carried out to measure the performance of the proxy server in terms of speedup, efficiency, scalability, and throughput (requests/sec). Our implementation is also contrasted with python's ayncio library, which provides a similar API as the proxy server to handle requests using multiple threads. The package bypasses python's global interpreter lock (GIL) by performing only IO-bound operations such as socket access. GIL is released for IO, allowing for non-blocking requests akin to the select() system call [9]. Asyncio is used in production at several companies, notably Facebook [10], making it a realistic choice as a parallelized benchmark for our proxy server.

The server simulation code is written in C and requires pThreads and libcurl libraries. Asyncio simulations and plotting code are written in python. All experiments were conducted on the NYU crunchy-1 server. Request sizes of 164,328, 656, and 1324 URL's were tested on the server using thread counts of 1, 2, 4, 8, 16, 32, and 64. Each URL is unique, meaning that there is both a read and a write to the web cache on every request. An expensive socket call (curl) will be made for each request since it will not have been previously cached. These two factors allow us to simulate heavy load on the server. A server with a partial filled cache will certainly handle more requests/sec than a server with no cache or one with an empty cache, purely because the number of expensive network requests is reduced. However, these simulations attempt to test the performance of the server and hash table under worst case scenarios, not the efficacy of the caching policy (which arises in nice scenarios where particular links are frequently requested).

At the end of the simulation, plots of speedup, efficiency, and throughput are displayed for our server and asyncio. In addition, simulations are performed for a version of the web cache using a CPU cache-misaligned hash table. This experiment will demonstrate the benefits of cache awareness.

RESULTS AND DISCUSSION

Speedup of our proxy server is quite good, with a slope > 1 for all input sizes until 16 threads in the pool (Figure 4). From then on, additional threads degrade performance. Asyncio exhibits sub-optimal speedups, with a slope < 1 for all input sizes. The gain in speedup is constant as the number of threads increases until 8 threads after which, performance degrades. This indicates our thread pool implementation for task management was efficient, exhibiting greater speedups for higher thread counts than asyncio.

Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Figure 3

The efficiency plots tell a similar story (Figure 5). Efficiency of asyncio drops sharply as the number of threads increases past 8, for all input sizes. The efficiency of our proxy server is either flat or increasing up till 8 threads, after which there is a slow drop in efficiency. However, our server nearly exhibits weak scalability. For example, as the number of requests doubles from 328 to 656 and the number of threads double from 32 to 64, efficiency changes only slightly from 0.81 to 0.79.

The throughput of our server is about 3X better than asyncio, 128 requests per/sec compared to 40 requests/sec when using 64 threads (Figure 6). Throughput follows a similar pattern as speedup i.e. throughput increases at a near constant rate till 16 threads, then diminishes. Diminishing speedup and throughput can possibly be explained by the use of read-write.

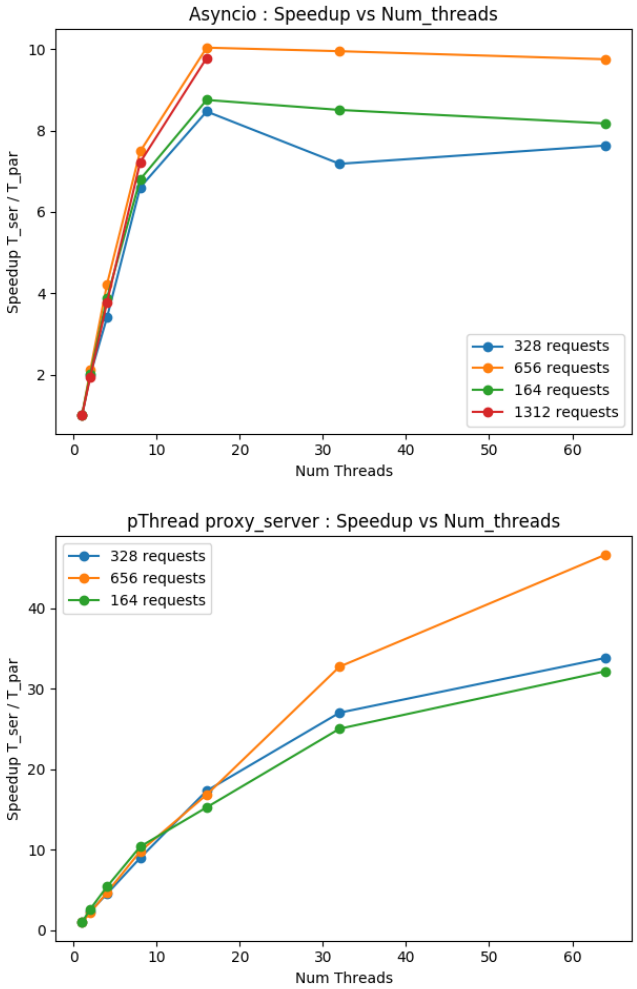
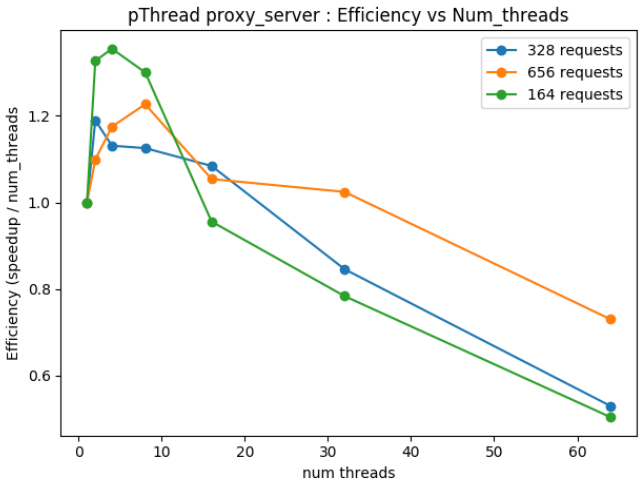


Figure 4

locks. Figure 3 displays the performance of different locking schemes in a linked list [11]. Read-write locks perform the best when a large percentage of requests are reads relative to writes. This is expected since reads are non-blocking. However, a single write will lock the entire table, leading to blocking of any threads that want to read. This can lead to a performance hit, especially in our situation where exactly half of all table accesses are reads and half are writes. This suggests that there is an optimal way to split the hash table such that a different lock is used for each partition of the table. In doing this, we avoid storing a lock for each entry. We also gain concurrency since entries in different table partitions can be written to simultaneously.



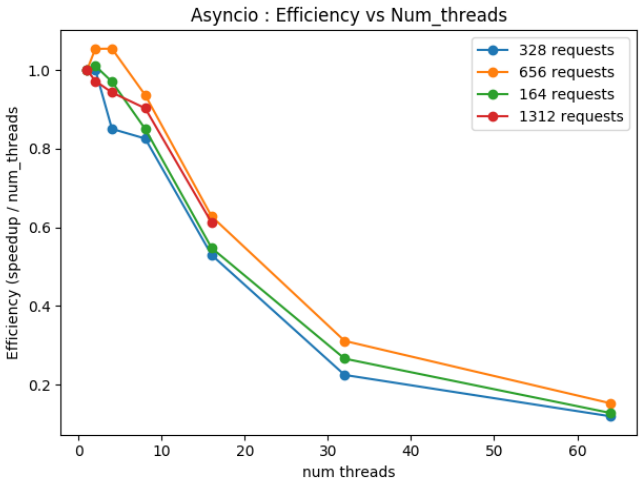
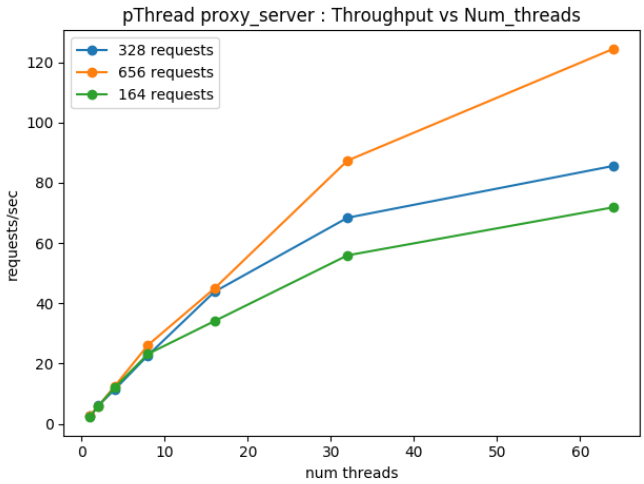


Figure 5



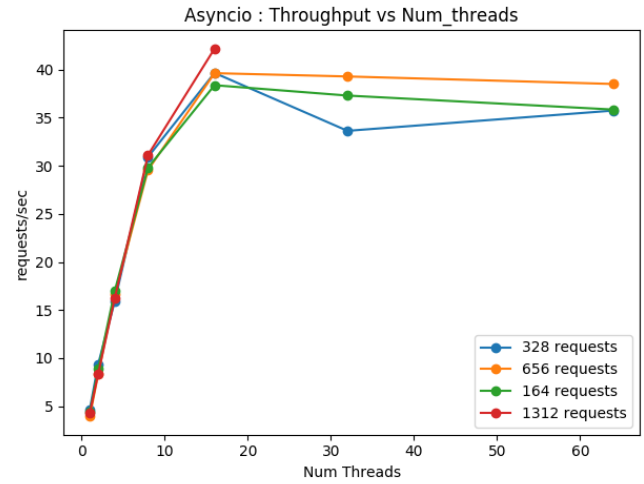


Figure 6

A peculiar anomaly in the plots is the fact that asyncio speedup and throughput suddenly flattens after 16 threads, instead of simply diminishing. This suggests an artifact inside the python library is restricting thread count, perhaps the GIL itself.

We demonstrate the benefit of a cache-aware hash table by artificially misaligning elements in the CPU cache and collecting performance metrics. This is done by adding an unused single character field to the hash table entry struct. The table below shows the effect of cache awareness on efficiency for multiple input sizes and a thread count of 16. A cache aligned hash table improves efficiency by ~ 9%.

(Input size)	Aligned	Mis-Aligned
328	1.1269	1.0268
656	1.5362	1.4168

NETWORK IMPLEMENTATION

This section describes a real-world network implementation of the proxy server. It also contains a web cache and is designed to handle up to 100 asynchronous clients launched on separate threads. Each client creates a TCP connection to the server and transmits a URL string. The server continuously listens for new clients and launches each one on a separate thread. It then takes the URL and implements the HTTP Get request via an external call to the libcurl library. The HTML file returned by cURL is then forwarded to the client in 1024 byte chunks. Likewise, the client also receives data in 1024 byte chunks, allowing the program to maintain buffer integrity. Once the client has the whole html file, it renders it in a browser via the bcat terminal

utility for user display. In addition to this, there are several enhancements:

- A backend MySQL database stores performance data on all requests, as well as the web cache. This allows the server to maintain several state variables and serves as a useful network management tool for the admin.
- If a particular client does not make a url request after a 300 second timeout, the client connection is closed by the server. This frees up threads and prevents us from hitting the 100-thread limit.
- A separate CRON job purges the cache every so often such that the web-pages stored are recent versions with the most up-to-date information.

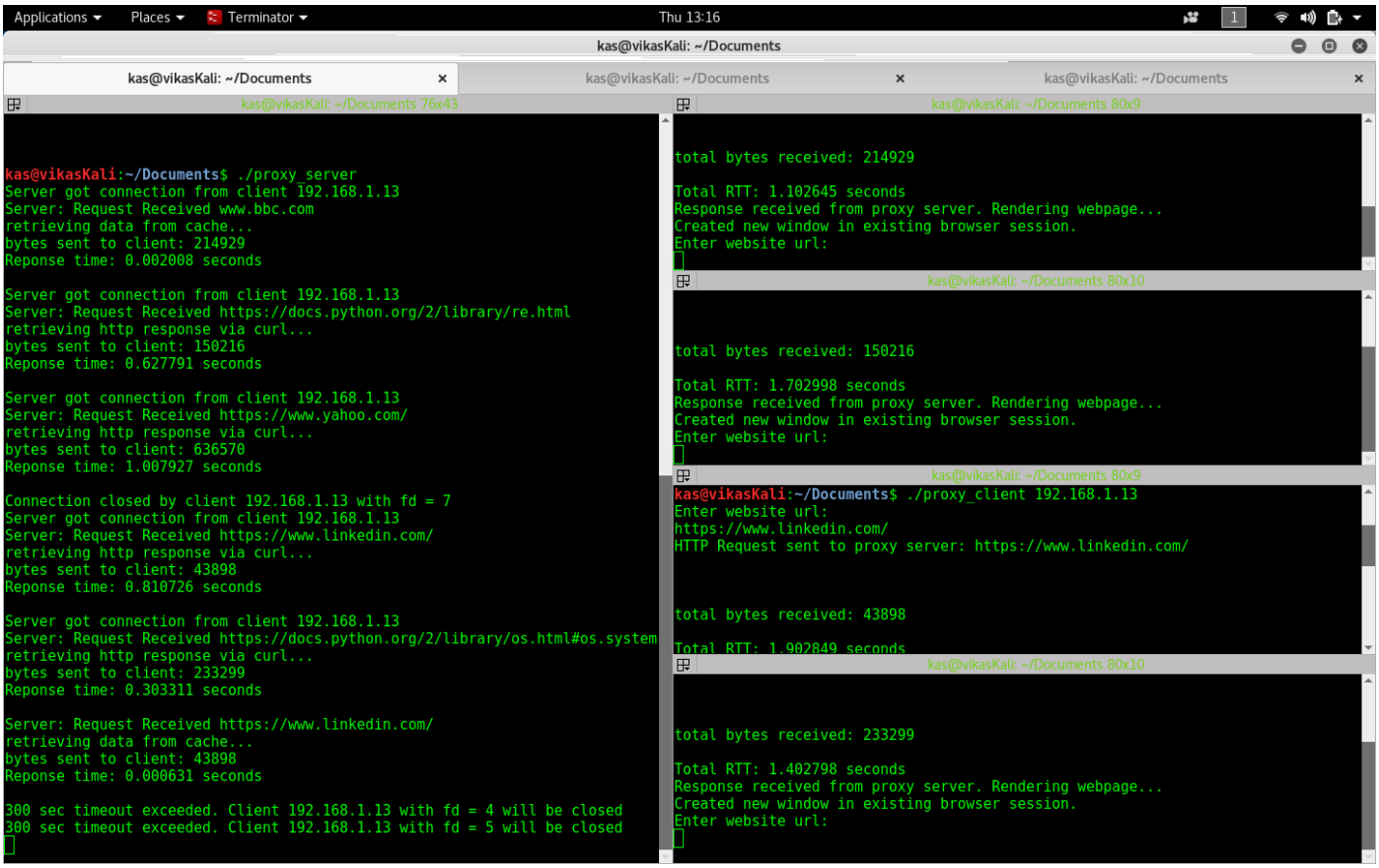


Figure 7 – Terminal display of the proxy server (left) and 4 clients (right)

CONCLUSION

This project can be summarized in the following points.

- Our new server performs reasonably well, approximately 128 requests/sec compared to asyncio's 40 requests/sec on 64 threads.
- Efficiency of asyncio falls sharply as the number of threads is increased. Our new server nearly exhibits weak scalability.
- Our new server achieves great speedup up to 16 threads but the gains from parallelism start diminishing with additional threads

REFERENCES

[1] Jonas Boner (2019) latency.txt source code (Version 2.0) [Source code]. https://gist.github.com/jboner/2841832 Accessed Wed May 15 12:21:26 EDT 2019

[2] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta. Experiences parallelizing a web server with OpenMP. In IWOMP, 2006.

[3] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, Zhi-Li Zhang. Making Content Caching Policies 'Smart' using the DeepCache Framework. In SIGCOMM CCR 2019

[4] Harshal Datir, Yogesh Gulhane, P.R. Deshmukh. Analysis and Performance Evaluation of Web Caching Algorithms. In IJEST 2011

[5] Vinit Kakde, Sanjay K.Mishra, Amit Sinhal, Mahendra Mahalle. Survey of Effective Web Cache Algorithms. In IJSEI 2012

[6] Zviad Metreveli, Nickolai Zeldovich, M. Frans Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In SIGPLAN 2012

[7] Rasmus Pagh, Zhewei Wei, Ke Yi, Qin Zhang. Cache-Oblivious Hashing. In PODS 2010

[8] aappleby (2019) snhasher source code [Source code]. https://github.com/aappleby/smhasher Accessed Wed May 15 12:21:52 EDT 2019

[9] Python Software Foundation (2019) Asyncio [Source Code, Documentation] https://docs.python.org/3/library/asyncio.html Accessed Wed May 15 12:22:47 EDT 2019

[10] Facebook Inc. (2016) Python in Production Engineering. [Documentation] https://code.fb.com/production-engineering/python-in-production-engineering/ Accessed Wed May 15 12:26:09 EDT 2019

[11] Pacheco, Peter (2011). *An Introduction to Parallel Programming*, 1st Edition. Morgan Kaufmann