Profiling Latency Critical Applications Using Tailbench

Abhishek Aditya BS
Department of Computer Science
PES University
Bengaluru, India
abhishek.aditya10@gmail.com

Supreeth G Kurpad
Department of Computer Science
PES University
Bengaluru, India
supreethkurpad@gmail.com

Vishal R
Department of Computer Science
PES University
Bengaluru, India
vishalramesh01@gmail.com

Subramaniam K V
Department of Computer Science
PES University
Bengaluru, India
subramaniamkv@pes.edu

***Abstract*—Client server architecture is an architectural style in which a set of files and data reside on the server and the client makes requests from the front end to access it through a network of computers. In such applications, the response time or latency of requests - the time taken by the server to respond to the client - becomes a crucial factor for determining the performance of the system. Among all the latencies, the paper focuses on the tail latencies - the ones that take the most time to process. In this paper, we present our research on a suite of such applications - Tailbench. Tailbench contains 8 latency critical applications - masstree, sphinx, img-dnn, specjbb, silo, moses, xapian and shore. The paper contains extensive profiling of tail latencies - 95th and 99th percentile - of these 8 applications on different parameters in terms of Memory performance, CPU performance, I/O performance and many other metrics.**

I. Introduction

With the evolution of chip design, computers have become increasingly fast with respect to processing power. The main bottleneck with most of the online services offered today is the network and the latency between the client and the server. Latency is the amount of time it takes for a packet of data to be read, transmitted, processed through the server and reach the client. When a client interacts with an online service - residing on a server, it exchanges information with it in the form of requests and responses. For most of these requests, the server is capable enough to respond to the client with acceptable latencies. Contrarily, a few of the applications take longer than the mean response time. These latencies contribute to the tail latency.

Presently, most of the processing power is dedicated to optimising the throughput of applications rather than optimising the tail latency. Therefore, most of the resources today are spent on optimising the average latency or throughput rather than the tail latencies. This has a negative effect on the tail as extended optimisation of throughput eventually leads to longer tail latencies and hence affects the performance of latency critical applications that require almost all of their request response pairs to be within a constrained range. The major challenge in identifying the tail is that there are a very few data points that represent the tail latencies by the very nature of the problem. Therefore it requires extensive statistical tools and metrics to be profiled well. Tailbench covers this and many other issues in other latency profiling applications. With the evolution of a wide variety of online services such as search engines, online translator services, speech assistants and other latency critical systems, it has become very essential to profile the tail latencies and optimize it.

With the evolution of high speed internet and 5G networks, analysing tail latencies and profiling them has become very crucial. Latency critical applications are applications which will fail to perform or perform erroneously when the communication and processing within the application does not happen with a latency that falls within a maximum threshold.. With such high speed networks, the number of real time audio applications, Voice over IP (VoIP) applications, Cloud computing services and IoT devices - belonging to the class of latency critical applications keeps growing. Therefore, such strict constraints on performance creates the need for maximum optimisation of the servers and extensive analysis of their behaviour especially at the 95th and 99th percentile latencies.

In this paper, we analyze each of the 8 applications selected in Tailbench in depth. With the advantage of these applications being modular, they are integrated with a central application called harness. This is responsible for collection of latency details and writing them to each of the files so that they can be used for further analysis. the metrics that were analysed are the following - mean latency, 95th percentile latency p95, 99th percentile latency p99, p95/mean and p99/mean latency. The following metrics are reported under various configurations of each of the applications. The configurations represent the variety of load situations and the stress situations that the server that runs these applications might be in. The different configurations include the server under high CPU stress, Memory Stress, Disk I/O stress depending on the type of application being profiled. A set of function call graphs have been generated to identify the most used functions in each of the applications. This forms the goal of the research conducted in this paper.

II. Related Works

Kasture et.al. in their paper [1] explains the reasons for choosing the 8 Tailbench applications and the robustness of the benchmarking suite. The applications are carefully selected to represent a good variety of applications which have latency critical requirements. Tailbench contains multiple environments and configurations in which the applications can be profiled. Applications can be configured to run on a single node or multiple nodes if need be. Numerous validation metrics have been established for benchmarking the applications. The entire suite is modular and represents consistent testing scenarios in a variety of load configurations through different applications. The central harness application gathers data from all the applications and writes information into files for latency analysis. Several randomisation techniques have been used to maintain relative order of the requests so that the tail can be analysed efficiently. Tailbench records three-fold latency information - Service Time: The time taken by the server to process the request, Queue times to record the time which the request spends in the queue waiting for other processes/ requests to complete, Sojourn times contain the time taken for the entire life cycle of the request-response pair. The authors after several simulations conclude that running the applications on integrated mode considerably reduces the effort/cost required to profile latencies in a simulated environment.

Walker et.al. in their paper [2] tried to understand the core of Sphinx with respect to its modularity and scalability to be implemented in other languages. Sphinx is a flexible speech recognition system. The underlying implementation is based on Hidden Markov Models. Hidden Markov Models is a statistical model in which the data is modeled as a Markov Process. It contains a set of hidden states and a set of observable outcomes. The earlier version of Sphinx is written in C++. The authors have come up with a way to optimize sphinx by integrating it with a Java implementation in the form of Sphinx 4. Sphinx 4 allows for a high degree of parallelism. Sphinx 4 uses algorithms such as the Viterbi algorithm for searching on the decoder process. It also allows for traditional search algorithms such as DFS and A* searching algorithms for searching. The language model in Sphinx 4 can be represented in a variety of formats such as lists of words, finite state grammars, ngram models and so on. The authors explain that the main reason for choosing Java for Sphinx-4 is because the JVM can run on various platforms and provide consistent results. The garbage collector in Java aids the development process and abstracts the developers away from memory related issues. In the end, the authors conclude that Sphinx 4 performs much better than its previous versions when it comes to speed and accuracy. Sphinx 4 covers the future scope of speech recognition leaving more room for research and profiling.

Suresh et.al. in their paper [3] prove that replica selection strategy in client server applications - where the client takes a decision on selecting one out of many replica nodes - plays a significant role on the tail latencies of online data stores. The authors prove that their improved version of such an algorithm called the C3 selection algorithm decreases the tail latencies and thereby increases the efficiency of latency critical applications. The C3 algorithm developed by the authors contains 2 major concepts. Firstly, it uses an efficient feedback mechanism from client and server nodes where the clients make use of a ranking procedure where they rank servers on the basis of their service times. Secondly, While this is done, it is made sure that such a mechanism does not cause a load imbalance on the servers. The authors claim that such an algorithm is applicable in applications which involve key value data stores. The core of the C3 algorithm lies in handling concurrent execution and punishing servers which put the request into longer queue times. While ranking the servers, the algorithm takes care not to overwhelm a single server which is ranked high. The authors implement C3 in a Cassandra database on an Amazon EC2 instance. Finally, the authors demonstrate that their solution improves Cassandra's tail latency. They prove that there is a three fold improvement in the 99th percentile latency and results in a throughput that improved by a factor of 50%.

Haque et.al. in their paper [4] understood the effect of parallelism and multicore processing and multithreading on the latency of applications such as search engines, recommendation engines or financing services on the internet. Even with the presence of multicore, multiprocessor servers, the biggest problem is efficiently using all the cores and processors to serve requests from the server. The obstacle in this regard is to identify parts of the requests that can be parallelised and identify which requests actually have to be parallelised in order to improve the tail latency. This creates a need for dynamically approach the task of parallelization. The authors showcase the FM (Few to Many) model to achieve the same. This algorithm uses the most popular requests and creates a mix of hardware and software level parallelism. This computed value is then used to load balance the servers and add active parallelism during runtime. As the processing time of the request increases, the degree of parallelism that is allocated to it also increases and thereby makes sure that all the parallelism resources on the servers aren't wasted away on short requests that don't affect the tail latencies as much as the others. The authors develop an FM scheduler to decide the right mix of hardware and software parallelism. This algorithm is evaluated on a set of different search engines, both commercial and open source on a server with 8 core Intel processors. This former is Bing while the latter is Lucene. It is seen that the FM algorithm reduces the tail latency by about 26% in Bing and 32% on Lucene. Therefore, the FM algorithm efficiently utilises the parallelism capabilities of the server, taking into account various load conditions with the right mix of Software and Hardware parallelization.

Kang et.al. in their paper [5] state that one of the main reasons for tail latencies is Memory management and Garbage Collection in storage systems. The author claims that at the tail, Garbage Collection worsens the tail latencies by about 100 times when compared to the mean. The authors suggest a reinforcement based Garbage Collection mechanism. The technique proposed is to actively handle key states from a set of potential states. It maps the most popular key states such that the complete states don't have to be managed on Flash Storage devices. Reinforcement based Garbage Collection implements a scheme for memory management which exploits inter-request times. It trains on the performance and load on the server to identify intervals of time in which it can perform partial garbage collection and thereby save time and processing resources at the tail. The algorithm models it such that it maximises the reward in the form of good response times. The authors conclude that their implementation of Garbage Collection significantly reduces the tail latencies and it is backed up by the 23% reduced tail latencies proven by experimental results. Thus it can be stated that Optimized Garbage Collections leads to reduced tail latency.

III. Experiment and Experimental Results

Latency in applications can be from network delay or congestion, slower disk reads, limited memory capacity and bandwidth issues, other user and operating system processes running in the background which will hog the CPU and cause the application to be run in a stressful environment with limited resources. In our experimental work, we focused on tail latency (95th and 99th percentile of the request-response time) in applications from various domains like Image Recognition, OLAP Databases, Speech Recognition, Search Engines, etc.

Some of the famous benchmark suites like CloudSuite, BigDataBench, DCBench, AMPLab Big Data Benchmark, Sirius, YCSB, are less diverse in terms of range of latency critical applications they offer, and have small number of applications which are hard to emulate or simulate.

TABLE II

TAILBENCH APPLICATIONS RUNTIME STATISTICS

Application Name	*CPU %*	*No. Of cores*	*Memory %*	*Resident Set Size (kB)*	*Minor Page Faults*	*Major Page Faults*	*Voluntary Context Switches*	*Involuntary Context Switches*
Imgdnn	557	6	0.23	151216 (147.67M)	3,83,667	0	9,73,634	784
Moses	731	8	1.178	772276 (754.17M)	8,37,768	0	8,64,013	1,29,801
Masstree	251	8	23.08	14838308 (14.1G)	41,41,293	0	47,19,943	6,352
Sphinx	3115	32	5.38	3486544 (3.22G)	6,53,53,952	0	4,54,172	1,33,276
Specjbb	256	8	9.38	6130312 (5.92G)	15,59,833	9	46,99,710	87
Shore	40	8	6.669	2174812 (2.07G)	5,51,676	0	3,98,752	3,275
Xapian	759	8	0.083	53624 (52M)	1,31,606	0	3,87,041	716
Silo	189	8	38.56	25149400 (23.98G)	2,16,78,564	0	2,80,948	784

CloudSuite offers very few tail-latencies applications, and the load testers used in CloudSuite have technical inaccuracies, leading to substantial errors in latency measurements. A multi-node setup is used in CloudSuite applications. While this design closely resembles the topology of scale-out systems, it makes it difficult to simulate tail latency correctly. BigDataBench offers only three latency-critical applications out of the nineteen applications it offers, and suffer from the same problem of multi-node setup, which lacks from a comprehensive approach for monitoring latencies.

Tailbench, a latency benchmark suite built solely to measure tail latencies in latency-critical applications, is chosen in our experiment. Tailbench offers applications from a wide range of domains in real world scenarios and all the applications offered are latency-critical with wider range of tail latencies from microseconds to seconds. All the applications offered are controlled by a central harness which logs the all the latency related information therefore providing reliable unbiased estimates. Tailbench offers both single-node and multi-node setup configurations, networked and integrated configurations to simulate real world scenarios.

A. Tailbench Applications

Tailbench provides 8 different applications from a wide range of domains. These applications are

- Xapian - a C++ based open-source search engine that is frequently utilised on major websites.
- Masstree - a C++-based in-memory key-value store that is efficient and scalable. For a wide range of services, in-memory key-value stores function as data storage backends.
- Moses - a C++ based statistical machine translation (SMT) system that is state-of-the-art. Language translation services like Google Translate rely on SMT algorithms, and speech-based interfaces like Apple Siri rely on them as well.
- Sphinx - C++ based efficient voice recognition system. Voice Recognition systems are a significant part of speech-based interfaces and apps like Apple Siri, Google, and Microsoft Cortana.
- Silo - a transactional database that runs in memory. Silo is built to scale efficiently on current multicores, eliminating centralised contention spots and maximising memory hierarchy usage.
- Shore - It is an on-disk database, unlike silo, it stores and accesses data in a very different way.
- Imgdnn - OpenCV based handwriting recognition program. Optical character recognition, picture-based search example Google Lens, automatic image tagging, and a number of other web applications are all examples of image recognition applications.
- Specjbb - a java middleware benchmark used widely in the industry and fir commercial services. It is frequently required to meet severe latency requirements.

B. Profiling all Tailbench applications

TABLE I

SYSTEM CONFIGURATION

System Configuration			
Operating System	***CPU***	***RAM***	***HDD***
Ubuntu 18.04 LTS	Intel Xeon 2.3Ghz	64 GB DDR4	1 TB

Table I shows the system configurations used to carry out the experiment. The Tailbench inputs are available at link. These are the datasets using which the models in the applications are trained. The directory structure with all the inputs, application models, and a scratch directory to write logs and other temporary files was set up. After successful compilation of all the applications following the instructions in each of the application's readme text files, the applications were run and the following statistics were recorded on each application shown in table II. Then the tail latency ratios (P95/μ and P99/μ) were recorded to determine the significance of latencies compared to the mean of the request-response time.

As seen from the table, we can clearly say which all applications are CPU intensive and which all are memory intensive by comparing the CPU utilisation and memory resident set size. ImgDnn, Moses, Sphinx, Xapian are CPU intensive applications utilising 99% of the CPU during its execution. MassTree, Specjbb, Shore, Silo are memory intensive applications because of its huge resident set size. By categorising applications as CPU vs memory intensive we can narrow down our optimisation techniques and look

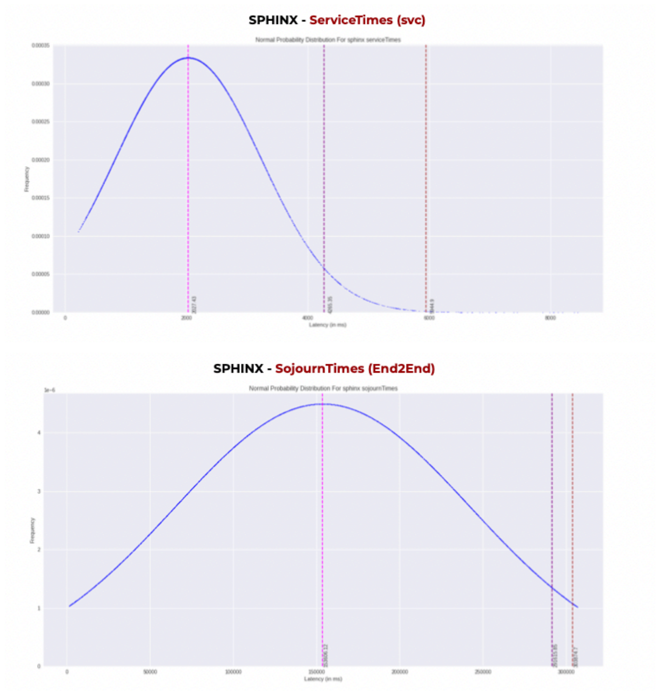

Fig. 1 Shows the SVC and End2End latency graphs for sphinx application.

for ways for improvement in latencies to either CPU domain or memory domain to further reduce the tail latencies.

TABLE III
TAILBENCH APPLICATIONS BENCHMARK LATENCIES

	SVC				*End2End*			
Application Name	*Mean μ (ms)*	*P95 (ms)*	*P99 (ms)*	*Max (ms)*	*Mean μ (ms)*	*P95 (ms)*	*P99 (ms)*	*Max (ms)*
Imgdnn	1.007	1.295	1.406	6.425	1.870	3.617	6.889	23.124
Moses	3.987	5.086	5.159	17.296	29869	56622	58986.33	59565.3
Masstree	0.609	0.688	0.750	14.425	433686	822406	856117.5	864748.5
Sphinx	2027.4	4265.4	5944.9	8446.5	153606	291615	303974.7	306933.3
Specjbb	0.073	0.087	0.294	269.9	59526	112958	117664.9	118840.9
Shore	3.129	2.401	91.935	1334.6	38660	75063	77601.9	78028.7
Xapian	0.967	2.438	2.833	4.253	29945	56773	59134.4	59729.2
Silo	0.319	0.923	1.339	7.140	10385	14887	15439.1	15587.1

Latencies of all the applications was measured. In table III detailed summary of SVC latencies (amount of time required by the server to receive a request and send a response) and End2end latencies (amount of time elapsed from the start of the request at the client till the response is received back to the client from the server) are shown. Here *P95* and *P99* indicates 95th percentile and 99th percentile of the request service time. Mean represents the 50th percentile and Max indicates the total time elapsed in the request response service.

By taking the ratios P95/μ, P99/μ we can measure the significance of the tail latency w.r.t to mean of the request-response time. Table IV shows these tail latency ratio for both SVC and End2End for all the applications. Sphinx, Xapian and Silo have high P95/μ, P99/μ ratios showing that they suffer from significant tail latencies. Results derived from table II, which shows Sphinx and Xapian which consumes 99% of the CPU and Silo which has very high resident set size are one of the reasons for such high tail latencies, because of the resource contention among the processes, thereby further increasing the delay in the request-response service time.

TABLE IV
TAIL LATENCY RATIO

	SVC		*End2End*	
Application Name	*Ratio (P95/μ)*	*Ratio (P95/μ)*	*Ratio (P95/μ)*	*Ratio (P95/μ)*
Imgdnn	1.285	1.396	1.934	3.683
Moses	1.275	1.384	1.895	1.99
Masstree	1.129	1.231	1.896	1.974
Sphinx	2.103	2.93	1.898	1.978
Specjbb	1.191	4.027	1.897	1.976
Shore	0.767	29.381	1.941	2.591
Xapian	2.521	2.929	1.895	1.974
Silo	2.893	4.197	1.433	1.486

Tail Latency graphs were plotted for all the applications and to understand the shape and extent of the tail. Fig. 1 shows the graph for sphinx service times (SVC) and the graph for sphinx end2end times (End2End). As it can be seen in the figures, the tail (right skewness) in the graph which shows that the tail latency is present and the extent of it is significant. If the tail latency was absent there should not be any right skewness in the distribution.

V. PROFILING SPHINX

Instead of profiling all the eight applications offered by the Tailbench benchmark suite, we decided to profile sphinx, extensively since Sphinx is a speech recognition system and a CPU intensive process consuming 99% of the CPU during its execution as shown by the table II. Due to increasing popularity, demand, wide range of use case scenarios of speech recognition systems, it's beneficial to understand the latency causes and to arrive at optimisation techniques to minimise the tail latencies in such applications.

It's essential to identify the dataflow and analyse how latency information propagates through the application. One way to identify this is to generate function call graphs and to analyse how much each function spends its time processing and trying to optimise the function which takes maximum amount of time. Function call graph was generated for sphinx using linux perf and valgrind tools. Both the graphs are available here. By stressing the server resources and understanding how the application performs is another important factor to understand the causes for the latency. We simulated six stress environments to measure how Sphinx performs to mimic the real world cloud servers handling billions of requests and requiring to respond to the requests within a fraction of a second. The six stress conditions are explained in the table V.

The stress simulations are done with linux tool stress-ng. Monitoring the system was done using analytical linux tools like htop, iostat, iotop, dstat, cachestat, Ftrace, Perf, etc. Caching in general increases the execution time of processes since the CPU doesn't have to wait for the DiskIO to be completed to fetch all the required pages of the processes and instead can fetch the process pages cached in the main memory. So all the stress simulations shown in table V are done with both inputs cached and not cached in the main memory of the system. This was done to check if caching of the inputs plays any role in the reduction of the tail latencies. Table VI shows the run time statistics of the sphinx execution (ran with 16 physical cores allocated for execution) under the above mentioned stress simulations.

TABLE V

STRESS ENVIRONMENTS DESCRIPTIONS

Stress Environment	***Description***
Baseline Sphinx	Sphinx run under no stress simulation
Baseline Sphinx with Inputs Cached	Sphinx run under no stress but with inputs cached in the main memory
100% CPU Stress, no caching	Sphinx run with 100% stressed CPU with no inputs cached.
100% CPU Stress with caching	Sphinx run with 100% stressed CPU with inputs cached in main memory
100% Memory Stress, no caching	Sphinx run with 100% stressed or filled memory with no inputs cached.
100% Memory Stress, with caching	Sphinx run with 100% stressed or filled memory with inputs cached.
Disk Stress, no caching	Sphinx run under stressed DiskIO with no inputs cached
Disk Stress with caching	Sphinx run under stressed DiskIO with inputs cached
All Stress enabled, no caching	Sphinx run under 100% stressed CPU, Memory and DiskIO with no inputs cached.

TABLE VI

RUNTIME STATISTICS OF SPHINX UNDER STRESS SIMULATIONS

Simulation Environment	***CPU %***	***Memory %***	***Minor Page Faults***	***Major Page Faults***	***Voluntary Context Switches***	***Involuntary Context Switches***
Baseline Sphinx	1543	2.641	9256940	84	169507	3363
Baseline Sphinx with Inputs Cached	1590	2.641	3753442	0	165736	1172
100% CPU Stress, no caching	756	2.634	12909349	67	250145	69377
100% CPU Stress with caching	755	2.663	10877392	0	319550	76696
100% Memory Stress, no caching	1007	2.619	9437300	1753	191338	10698
100% Memory Stress, with caching	1070	2.623	9448113	1265	175121	14856
Disk Stress, no caching	1256	2.644	8175125	144	169569	4651
Disk Stress with caching	1439	2.649	5654889	0	166160	2816
All Stress enabled, no caching	276	5.222	13687386	661	872308	159402

It can be seen from the table VI, both the baseline sphinx (sphinx run with no stress simulation) with or without inputs cached utilise the same amount of memory and CPU. But the difference is in minor and major page faults, sphinx with cached inputs have 0 major page faults and 1/3rd of minor page faults of the sphinx without inputs cached. With 100% CPU stress, with or without cache sphinx perform the same but their CPU utilisation drops to half of the CPU available for its execution. Since Sphinx is a CPU intensive application the above result was expected. With memory stress both sphinx perform the same but have very high major page faults. DiskIO stress shows sphinx with inputs cached has half the involuntary context switches compared to the sphinx with no inputs cached. With all the stress simulations enabled sphinx was able to utilise only 17% of the CPU, double the amount of memory usage compared to other sphinx executions, high major and minor page faults, as well as high voluntary and involuntary context switches. Detail analysis of sphinx with latencies measured under the stress simulations is shown in table VII.

It can be seen from table VII sphinx with 100% CPU stress has very high P95 and P99 SVC latency (in ms) almost twice the latency compared to the baseline sphinx with no stress simulation. When all the stress simulations are enabled the P95 and P99 SVC latencies are almost 4.5 times compared to the baseline sphinx. End2End latencies also show similar behaviour as SVC but End2End is not of interest to us because End2End time can be high due to network related issues like network failure or network congestion. But since SVC time only measures the duration between the receipt of the request by the server from the client to the time of response from the server, so its of practical significance since it measures the actual latencies caused due to the server or system issues like resource contention, slower DiskIO, etc.

TABLE VII

SPHINX IN DEPTH ANALYSIS UNDER STRESS SIMULATIONS

	SVC				***End2End***			
Simulati - on	***Mean μ (ms)***	***P95 (ms)***	***P99 (ms)***	***Max (ms)***	***Mean μ (ms)***	***P95 (ms)***	***P99 (ms)***	***Max (ms)***
Baseline Sphinx	1153.4	2501.8	3287.1	4055.8	32272.2	59762.9	62480.1	63270.0
Baseline Sphinx with Inputs Cached	1094.1	2276.8	3079.206	4261.249	30345.717	55864.2	58407.7	59059.1
100% CPU Stress, no caching	2209.9	4770.2	5882.7	8900.2	64530.6	121956.4	127193.3	128966.4
100% CPU Stress with caching	2224.8	4709.5	6115.7	8399.1	65111.3	122566.2	128125.6	129633.4
100% Memory Stress, no caching	1184.5	2559.1	3357.8	4234.6	34357.5	62775.4	65503.5	66237.1
100% Memory Stress, with caching	1181.3	2511.5	3241.5	5184.1	35434.5	64129.9	66829.9	67465.1
Disk Stress, no caching	1130.8	2338.3	3061.9	4576.7	34340.7	63343.1	66022.2	66845.1
Disk Stress with caching	1162.155	2478.8	3177.2	5075.7	32356.3	60062.8	62491.0	63349.9
All Stress enabled, no caching	3835.6	9401.7	13527.1	17813.7	63036.7	108691.4	112937.951	113992

TABLE VIII

TAIL LATENCY RATIO OF SPHINX UNDER STRESS SIMULATIONS

	SVC		***End2End***	
Simulati- on	***Ratio (P95/μ)***	***Ratio (P95/μ)***	***Ratio (P95/μ)***	***Ratio (P95/μ)***
Baseline Sphinx	2.169	2.850	1.852	1.936
Baseline Sphinx with Inputs Cached	2.081	2.815	1.841	1.925
100% CPU Stress, no caching	2.158	2.662	1.890	1.971
100% CPU Stress with caching	2.117	2.749	1.882	1.968
100% Memory Stress, no caching	2.160	2.835	1.827	1.907
100% Memory Stress, with caching	2.126	2.744	1.810	1.886
Disk Stress, no caching	2.068	2.708	1.845	1.923
Disk Stress with caching	2.133	2.734	1.856	1.931
All Stress enabled, no caching	2.451	3.527	1.724	1.792

From table VIII, it can be seen that sphinx run with 100% CPU stress and with all stress simulations enabled have high SVC P95/μ and P99/μ ratios indicating the presence of tail latencies due to right skewed distribution.

Since Sphinx utilises all the cores of CPU, therefore running it under CPU or all the stress enabled makes it wait for the CPU cycles, causing high latencies. In order to bring down the latencies techniques like optimisations in CPU scheduling, efficient CPU contention handling, effective and efficient multi threading and multi core utilisations can be researched and employed. Another way to minimise the latencies is by having more nodes and having a cluster of nodes with distributed computing with proper load and stress balancing amongst the nodes, but this comes at a huge cost, high maintenance and robust fault tolerance system.

VI. FUTURE WORKS

To profile the system along the life cycle of a request/ response. Once this is achieved, we will be able to map the system state to the latency that is observed, which will better help us pinpoint to the root cause of the tail latencies. Since Sphinx is a relatively old application which uses HMM based models for training and prediction, it cannot be used for training with complex and noisy data, and cannot train and predict once the data becomes huge for the model to handle. Kaldi, another relatively new and faster speech recognition application written in c++ which uses neural network models which introduces non linearity by its activation functions and hence can capture more complex data information to train. Kaldi has proven to work well with huge training set size and faster than existing speech recognition models. So by profiling and understanding how latency propagates through the system in Kaldi, we can look for optimisation techniques to reduce these latencies and hence reduce delay in request-response time, for users accessing Kaldi.

VII. CONCLUSION

Sphinx is a CPU intensive process and can be configured to use 100% of all the 32 cores on the server. The mean of the response times for the requests increases on increasing the stress on the CPU and memory. The mean response time remains unchanged irrespective of the number of page faults. Caching is not a major factor for the performance of the application. Although the mean response time for the requests increases, the ratio of the 95th and 99th percentile to the mean remains approximately the same.

REFERENCES

1. Kasture, Harshad, and Daniel Sanchez. "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications." In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1-10. IEEE, 2016.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
2. Walker, Willie, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. "Sphinx-4: A flexible open source framework for speech recognition." (2004).K. Elissa, "Title of paper if known," unpublished.
3. Suresh, Lalith, Marco Canini, Stefan Schmid, and Anja Feldmann. "C3: Cutting tail latency in cloud data stores via adaptive replica selection." In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 513-527. 2015.Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
4. Haque, Md E., Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. "Few-to-many: Incremental parallelism for reducing tail latency in interactive services." *ACM SIGPLAN Notices* 50, no. 4 (2015): 161-175.
5. Kang, Wonkyung, and Sungjoo Yoo. "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in SSD." In *Proceedings of the 55th Annual Design Automation Conference*, pp. 1-6. 2018.