Final Mini Project Demonstration
CCBD

Project Title	:	Tailbench
Project Guide	:	Dr. K.V. Subramaniam

Project Team (Name & SRN)
PES1UG19CS019 : Abhishek Aditya BS
PES1UG19CS520 : Supreeth G Kurpad
PES1UG19CS571 : Vishal R

Project Abstract and Scope

Tailbench is a benchmark suite and evaluation methodology for latency-critical applications. Tailbench consists of 9 different applications that can be used to obtain the tail latencies. The 9 applications are harness, img-dnn, masstree, moses, shore, silo, specjbb, sphinx, xapian.

Out of all the applications, we have limited the scope of the project to a single application - Sphinx.

Sphinx is a speech recognition system written in C++. Speech recognition systems are an important component of speech-based interfaces and applications such as Apple Siri, Google Now, and IBM Speech to Text. Sphinx performs speech recognition using random instances from the CMU AN4 alphanumeric database.

Kaldi is a state-of-the-art automatic speech recognition (ASR) toolkit and contains a wide variety of algorithms used in Automatic Speech Recognition systems. It also allows the user to train their own acoustic models on popular speech data such as the Wall Street Journal Corpus and TIMIT for example.

Project Abstract and Scope

What are acoustic models? Acoustic models are the statistical representations of a phoneme acoustic information. A phoneme here represents a member of the set of speech sounds in a language.

The acoustic models are created by training the models on acoustic features from labeled data, such as the Wall Street Journal Corpus, TIMIT, or any other transcribed speech corpus. Acoustic models are necessary not only for automatic speech recognition, but also for forced alignment. Kaldi provides tremendous flexibility and power in training your own acoustic models and forced alignment system.

Advantages of Kaldi over other toolkits

- Extensive linear algebra support
- Extensible design
- Available under open license
- Complete recipes
- The recipes have been designed in such a way that in principle they should never fail in a catastrophic way.
- Kaldi code is thoroughly tested.
- Kaldi code is easy to reuse and refactor, and easy to understand

Literature Survey

- 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.
Link : http://people.csail.mit.edu/sanchez/papers/2016.tailbench.iiswc.pdf

- 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.
Link : https://www.cs.brandeis.edu/~cs136a/CS136a_docs/Sphinx_Sun_tr-2004-139.pdf

- 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].
Link : https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-suresh.pdf

- 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.
Link : https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/FINAL-asplos168-haqueA.pdf

- 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.
Link :https://picture.iczhiku.com/resource/ieee/wyITdLTehTkjDxcX.pdf

Literature Survey

- Povey, Daniel, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann et al. "The Kaldi speech recognition toolkit." In IEEE 2011 workshop on automatic speech recognition and understanding, no. CONF. IEEE Signal Processing Society, 2011.
  Link : https://infoscience.epfl.ch/record/192584/files/Povey_ASRU2011_2011.pdf

- From the authors of Tailbench, the first paper 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.

- Second paper was reviewed 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.

- The third paper proves 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.

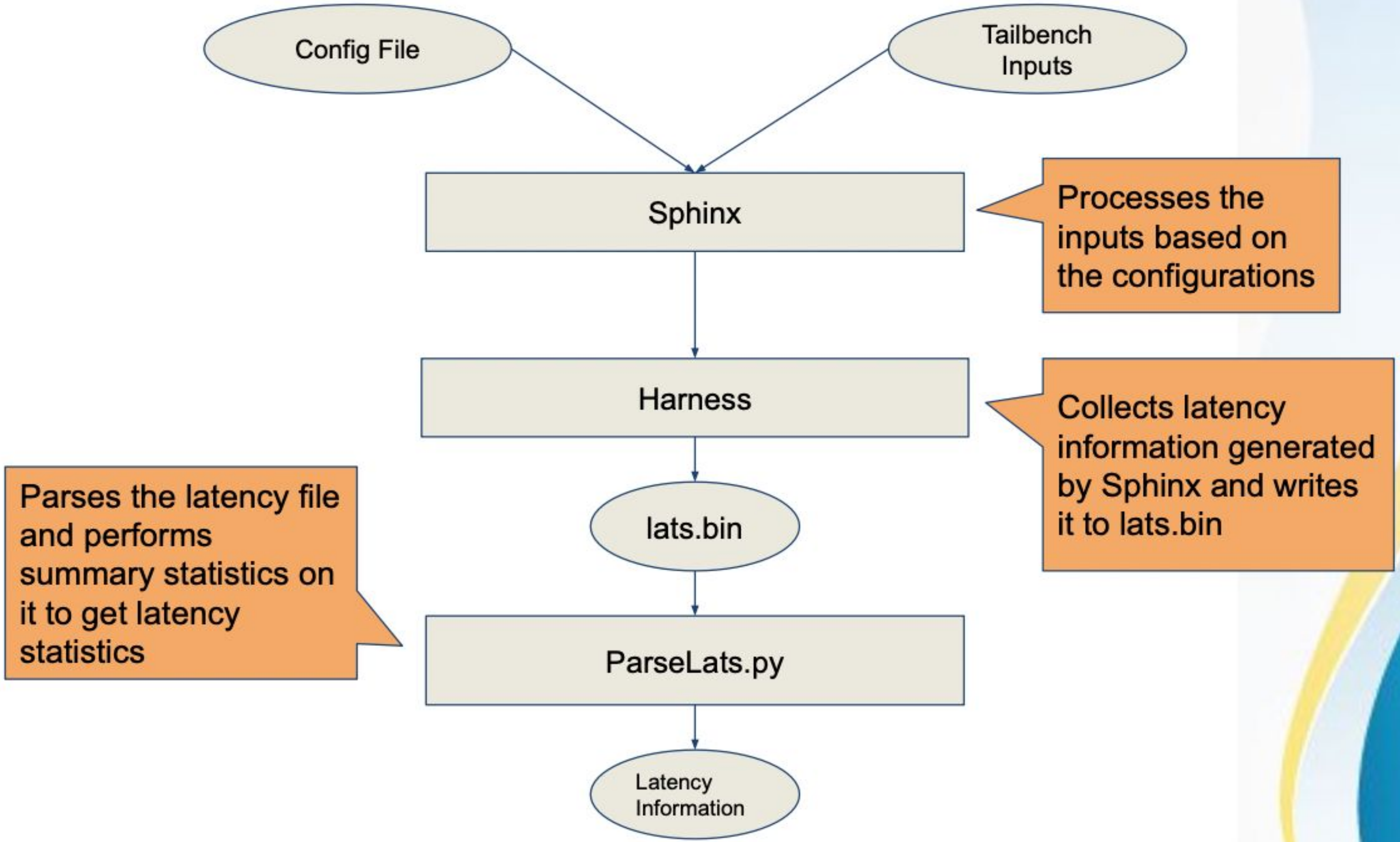
Literature Survey

- The fourth paper showcases the FM (Few to Many) model to optimize tail latencies. 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.

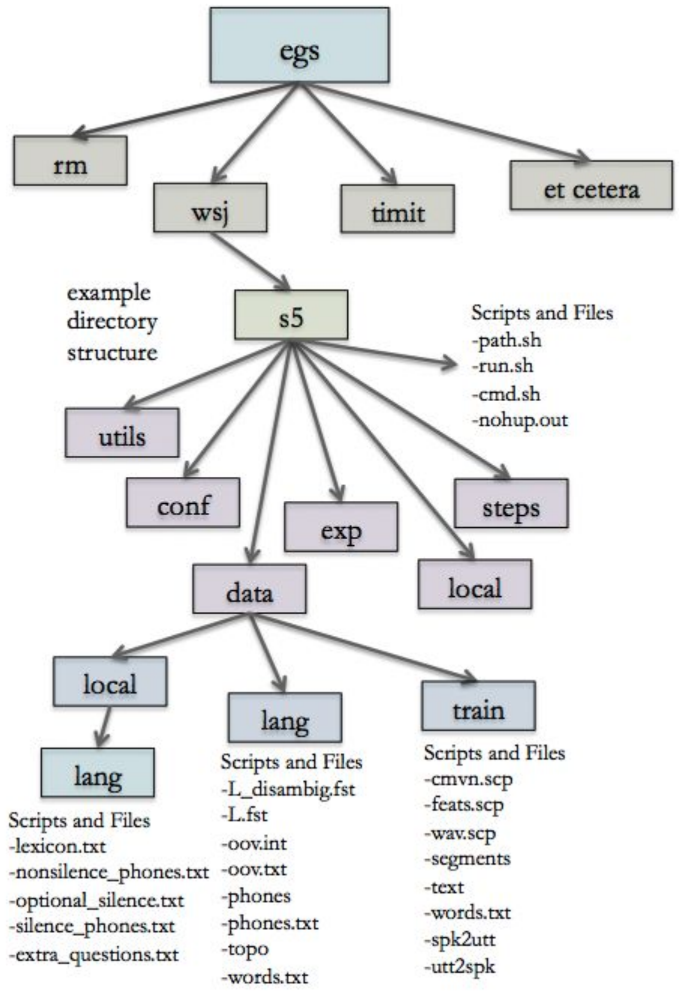
- The fifth paper states that one of the main reasons for tail latencies is Memory management and Garbage Collection in storage systems. The author claims that at the tail, Garbage Collection worsens the tail latencies by about 100 times when compared to the mean. The authors suggest a reinforcement based Garbage Collection mechanism. 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.

- Kaldi is a toolkit for Automatic Speech Recognition written in C++. Kaldi uses Finite State Transducers to achieve ASR. Kaldi also contains scripts for building complete ASR systems with support for arbitrary phonetic-context sizes and Gaussian Mixture Models. Kaldi eases the development process for a wide range of users.

System Design Of Sphinx


Config File
Tailbench Inputs
Sphinx
Processes the inputs based on the configurations
Harness
Collects latency information generated by Sphinx and writes it to lats.bin
lats.bin
Parses the latency file and performs summary statistics on it to get latency statistics
ParseLats.py
Latency Information

Design of Kaldi



- The top-level directories are egs, src, tools, misc, and windows. The directories we will be using are egs and src Training recipes are available for the Wall Street Journal Corpus (wsj), TIMIT (timit), Resource Management (rm), and many others. Under each of these directories are usually a few different versions.

- The highest number, usually s5, is the most current version and should be used for any new development or training.

- src stands for 'source' or 'source code' and contains most of the source code for programs that the training recipes call.

- For each training recipe directory, there is a standard sub-directory structure. The top directory contains the run script as well as two other required scripts. The sub-directories will primarily be using are data and exp.

- The exp directory will eventually contain the output of the training and alignment scripts, or the acoustic models.

Tailbench Context

- Tailbench contains 8 latency critical applications - masstree, sphinx, img-dnn, specjbb, silo, moses, xapian and shore.

- 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.

- We present 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. The central harness application gathers data from all the applications and writes information into files for latency analysis.

- 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 to complete.
  - Sojourn times contain the time taken for the entire life cycle of the request-response pair.

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
IMG-DNN	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	23.03	**14838308** (14.1G)	41,41,293	0	47,19,943	6,352
SPHINX	**3115**	32	5.38	3486544 (3.32G)	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	-	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

IMG-DNN : CPU Intensive
MOSES : CPU Intensive
SPHINX : CPU Intensive
XAPIAN : CPU Intensive

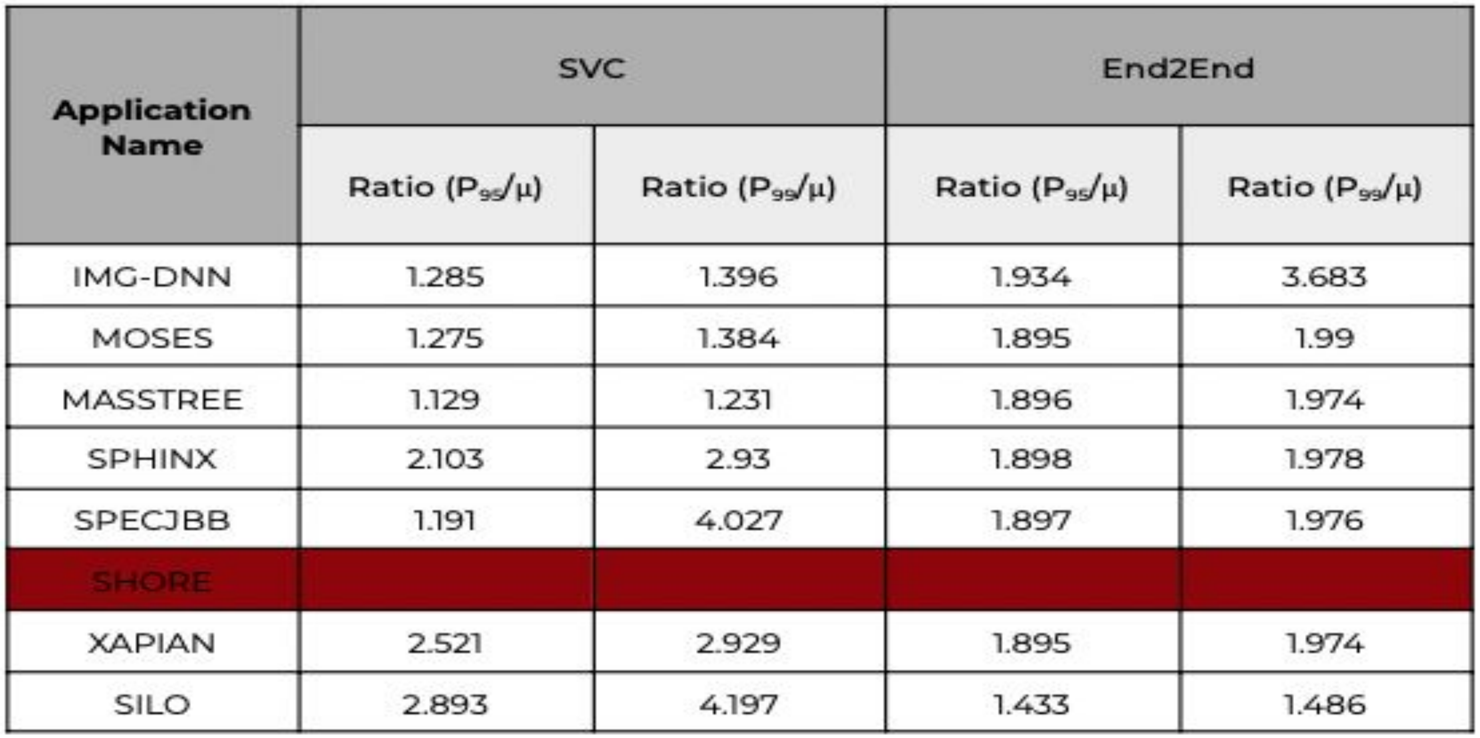
MASSTREE : Memory Intensive
SILO : Memory Intensive
SPECJBB : Memory Intensive

Tailbench Applications Benchmark Latencies

Application Name	SVC				End2End			
	Mean µ (ms)	P_{95} (ms)	P_{99} (ms)	Max (ms)	Mean µ (ms)	P_{95} (ms)	P_{99} (ms)	Max (ms)
IMG-DNN	1.007	1.295	1.406	6.245	1.870	3.617	6.889	23.124
MOSES	3.987	5.086	5.519	17.296	29869.017	56622.027	58986.33	59565.257
MASSTREE	0.609	0.688	0.750	14.425	433686.03	822406.97	856117.483	864748.48
SPHINX	2027.427	4265.354	5944.899	8446.489	153606.124	291615.852	303874.695	306933.296
SPECJBB	0.073	0.087	0.294	269.951	59526.287	112958.847	117664.926	118840.895
SHORE	3.129	2.401	91.935	1334.576	38660.457	75063.265	77601.917	78028.770
XAPIAN	0.967	2.438	2.833	4.253	29945.815	56773.292	59134.417	59729.240
SILO	0.319	0.923	1.339	7.140	10385.458	14887.983	15439.134	15587.182

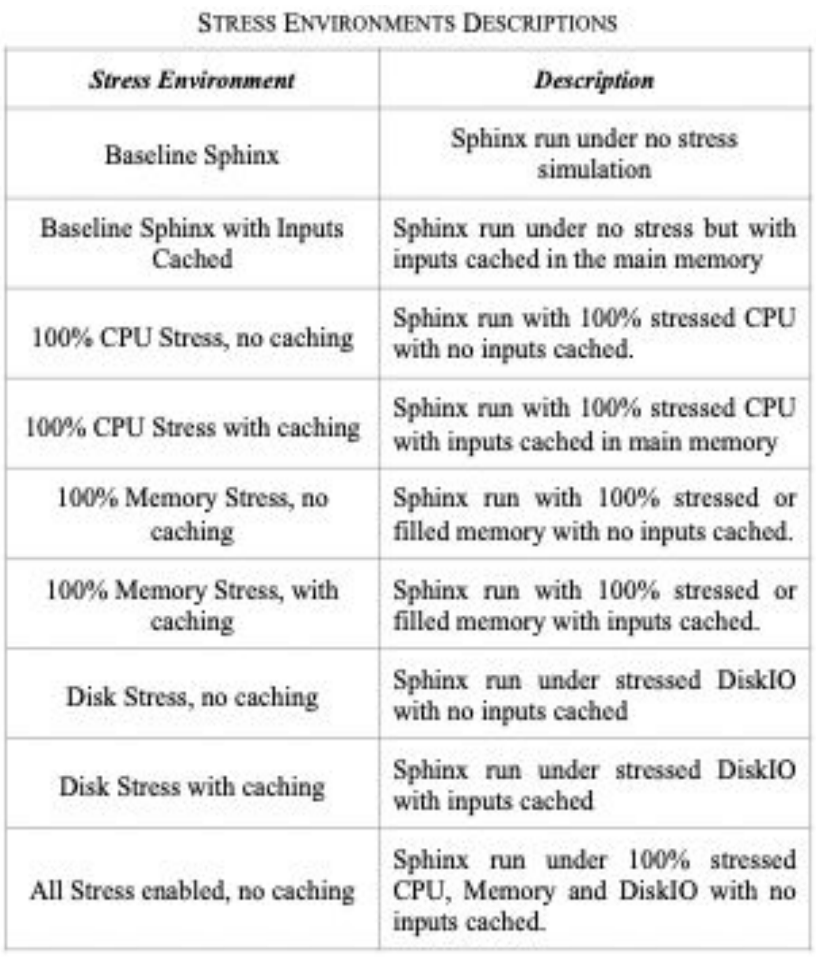
RED = Not Updated / Not executed / Not working

Tail Latency Ratio



Application Name	SVC		End2End	
	Ratio (P_{95}/μ)	Ratio (P_{99}/μ)	Ratio (P_{95}/μ)	Ratio (P_{99}/μ)
IMG-DNN	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				
XAPIAN	2.521	2.929	1.895	1.974
SILO	2.893	4.197	1.433	1.486

Sphinx Description



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.

- Instead of profiling all the eight applications offered by the Tail Bench benchmark suite, we decided to profile sphinx extensively since Sphinx is a speech recognition system and a CPU intensive process.

- Due to increasing popularity it's beneficial to understand the latency causes and to arrive at optimisation techniques to minimise the tail latencies in such applications.

- 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.

Sphinx run time statistics (No. of cores - 16)

Simulation	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	2.633	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
With Disk Stress, no Caching	1256	2.644	8175125	144	169569	4651
With Disk Stress, with Caching	1439	2.649	5654889	0	166160	2816
All Stress enabled, no caching	276	5.222	13687386	661	872308	159402

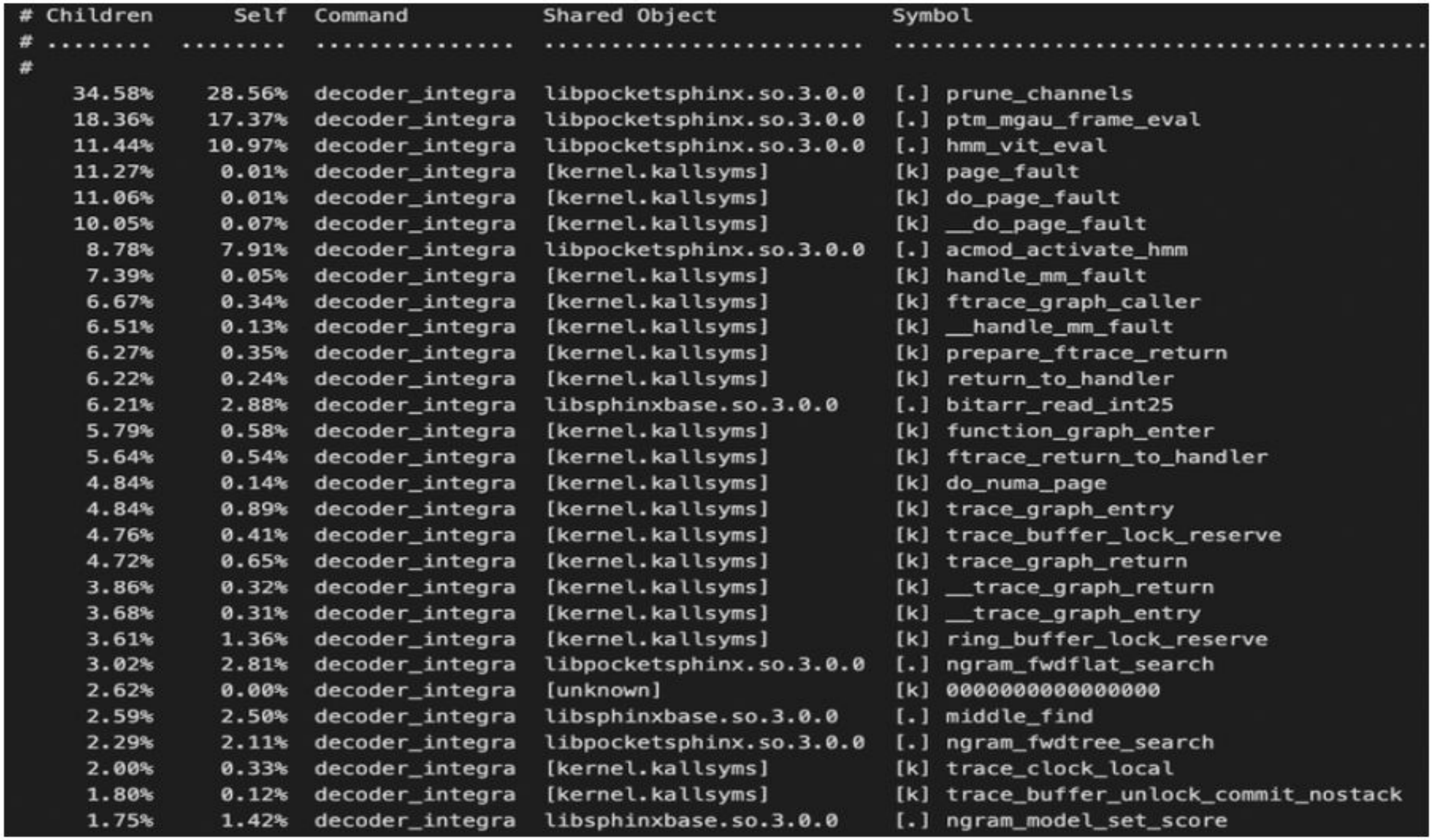
Sphinx In Depth Analysis Table

Simulation	SVC				End2End			
	Mean μ (ms)	P_{95} (ms)	P_{99} (ms)	Max (ms)	Mean μ (ms)	P_{95} (ms)	P_{99} (ms)	Max (ms)
Baseline Sphinx	1153.480	2501.867	3287.053	4055.808	32272.270	59762.979	62480.157	63270.994
Baseline Sphinx With Inputs Cached	1094.047	2276.824	3079.206	4261.249	30345.717	55864.293	58407.772	59059.081
100% CPU Stress, no Caching	2209.998	4770.250	5882.759	8900.264	64530.664	121956.424	127193.320	128966.486
100% CPU Stress, with Caching	2224.816	4709.530	6115.799	8399.061	65111.344	122566.199	128125.660	129633.443
100% Memory Stress, no Caching	1184.538	2559.141	3357.898	4234.687	34357.529	62775.426	65503.540	66237.082
100% Memory Stress, with Caching	1181.361	2511.504	3241.546	5184.172	35434.577	64129.922	66829.971	67465.058
With Disk Stress, no Caching	1130.883	2338.379	3061.872	4576.734	34340.758	63343.126	66022.271	66845.079
With Disk Stress, with Caching	1162.155	2478.815	3177.210	5075.731	32356.380	60062.856	62491.039	63349.906
All Stress enabled, no caching	3835.681	9401.788	13527.039	17813.691	63036.649	108691.447	112937.951	113992.645

Sphinx Tail Latency Ratio

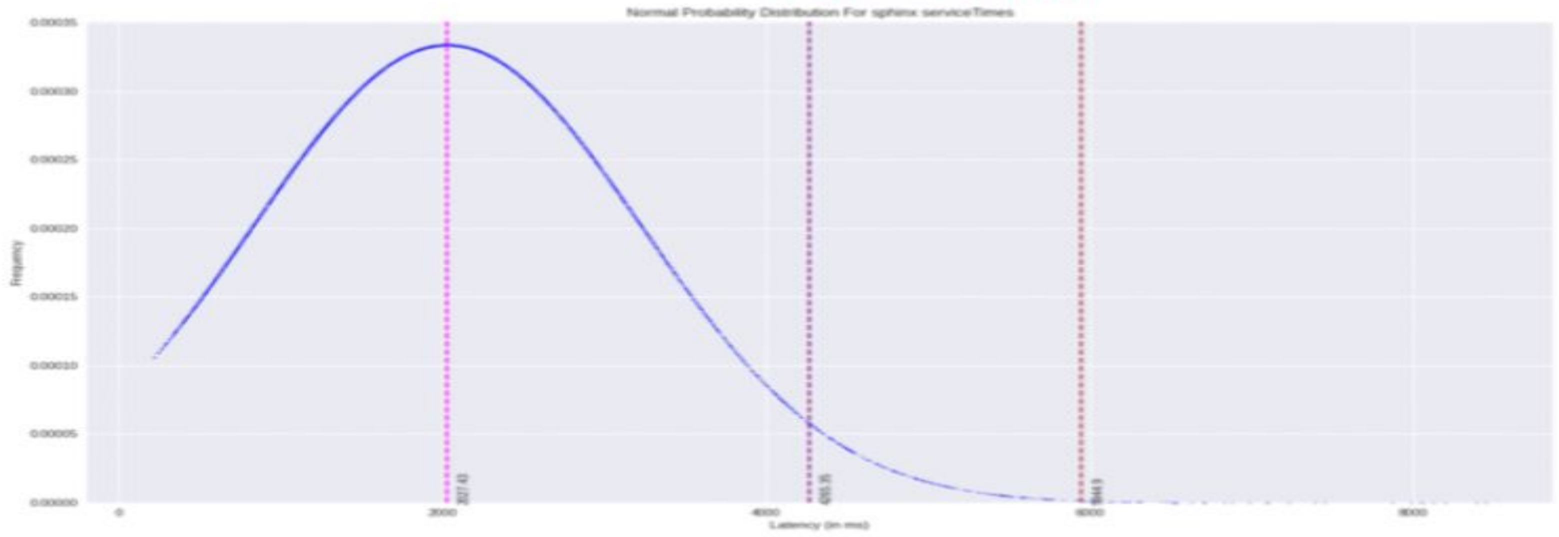
Simulation	SVC		End2End	
	Ratio (P95/μ)	Ratio (P99/μ)	Ratio (P95/μ)	Ratio (P99/μ)
Baseline Sphinx	2.169	2.850	1.852	1.936
Baseline Sphinx With Inputs Cached	2.081	2.815	1.841	1.925
100% CPU Stress, no Caching	2.158	2.662	1.890	1.971
100% CPU Stress, with Caching	2.117	2.749	1.882	1.968
100% Memory Stress, no Caching	2.160	2.835	1.827	1.907
100% Memory Stress, with Caching	2.126	2.744	1.810	1.886
With Disk Stress, no Caching	2.068	2.708	1.845	1.923
With 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

Function call statistics during the execution of sphinx



```
# Children      Self  Command          Shared Object             Symbol
# ........  ........  ...............  ........................  ........................................
#
    34.58%    28.56%  decoder_integra  libpocketsphinx.so.3.0.0  [.] prune_channels
    18.36%    17.37%  decoder_integra  libpocketsphinx.so.3.0.0  [.] ptm_mgau_frame_eval
    11.44%    10.97%  decoder_integra  libpocketsphinx.so.3.0.0  [.] hmm_vit_eval
    11.27%     0.01%  decoder_integra  [kernel.kallsyms]         [k] page_fault
    11.06%     0.01%  decoder_integra  [kernel.kallsyms]         [k] do_page_fault
    10.05%     0.07%  decoder_integra  [kernel.kallsyms]         [k] __do_page_fault
     8.78%     7.91%  decoder_integra  libpocketsphinx.so.3.0.0  [.] acmod_activate_hmm
     7.39%     0.05%  decoder_integra  [kernel.kallsyms]         [k] handle_mm_fault
     6.67%     0.34%  decoder_integra  [kernel.kallsyms]         [k] ftrace_graph_caller
     6.51%     0.13%  decoder_integra  [kernel.kallsyms]         [k] __handle_mm_fault
     6.27%     0.35%  decoder_integra  [kernel.kallsyms]         [k] prepare_ftrace_return
     6.22%     0.24%  decoder_integra  [kernel.kallsyms]         [k] return_to_handler
     6.21%     2.88%  decoder_integra  libsphinxbase.so.3.0.0    [.] bitarr_read_int25
     5.79%     0.58%  decoder_integra  [kernel.kallsyms]         [k] function_graph_enter
     5.64%     0.54%  decoder_integra  [kernel.kallsyms]         [k] ftrace_return_to_handler
     4.84%     0.14%  decoder_integra  [kernel.kallsyms]         [k] do_numa_page
     4.84%     0.89%  decoder_integra  [kernel.kallsyms]         [k] trace_graph_entry
     4.76%     0.41%  decoder_integra  [kernel.kallsyms]         [k] trace_buffer_lock_reserve
     4.72%     0.65%  decoder_integra  [kernel.kallsyms]         [k] trace_graph_return
     3.86%     0.32%  decoder_integra  [kernel.kallsyms]         [k] __trace_graph_return
     3.68%     0.31%  decoder_integra  [kernel.kallsyms]         [k] __trace_graph_entry
     3.61%     1.36%  decoder_integra  [kernel.kallsyms]         [k] ring_buffer_lock_reserve
     3.02%     2.81%  decoder_integra  libpocketsphinx.so.3.0.0  [.] ngram_fwdflat_search
     2.62%     0.00%  decoder_integra  [unknown]                 [k] 0000000000000000
     2.59%     2.50%  decoder_integra  libsphinxbase.so.3.0.0    [.] middle_find
     2.29%     2.11%  decoder_integra  libpocketsphinx.so.3.0.0  [.] ngram_fwdtree_search
     2.00%     0.33%  decoder_integra  [kernel.kallsyms]         [k] trace_clock_local
     1.80%     0.12%  decoder_integra  [kernel.kallsyms]         [k] trace_buffer_unlock_commit_nostack
     1.75%     1.42%  decoder_integra  libsphinxbase.so.3.0.0    [.] ngram_model_set_score
```

SPHINX - ServiceTimes (svc)



SPHINX - SojournTimes (End2End)

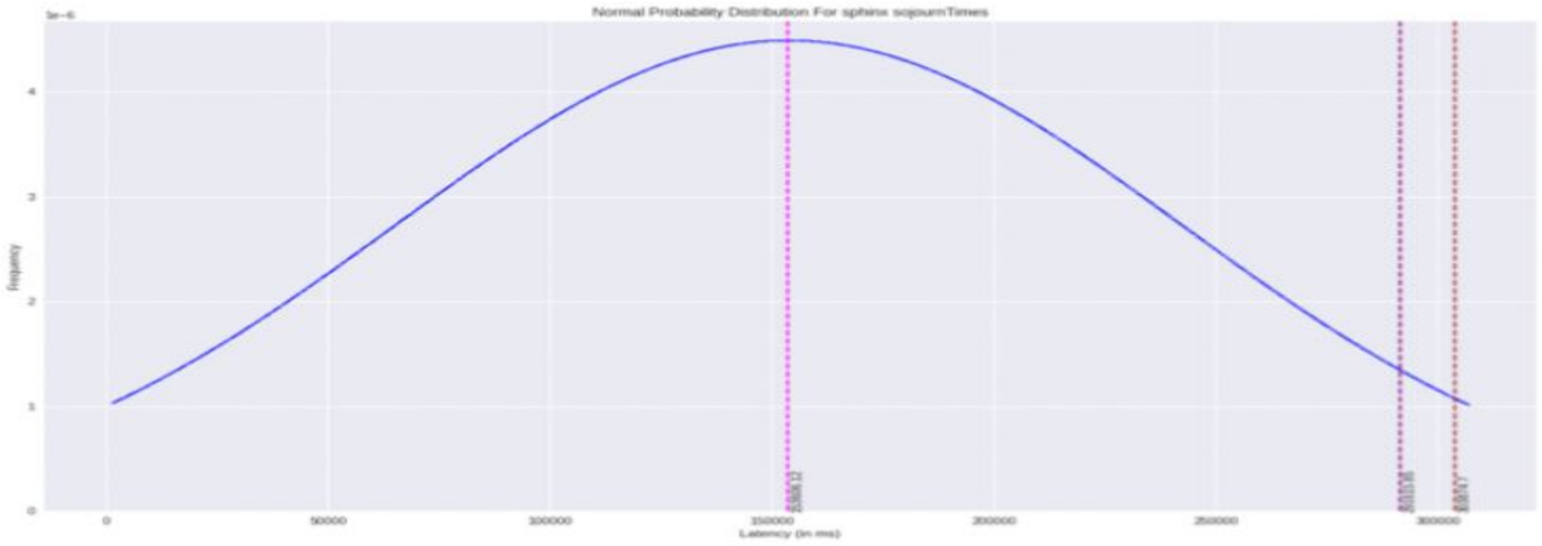


Fig. 1 Shows the SVC and End2End latency graphs for sphinx application.

Kaldi Training Overview

The procedure can be laid out as follows:

1. Obtain a written transcript of the speech data

   For a more precise alignment, utterance (~sentence) level start and end times are helpful, but not necessary.

2. Format transcripts for Kaldi

   Kaldi requires various formats of the transcripts for acoustic model training. You'll need the start and end times of each utterance, the speaker ID of each utterance, and a list of all words and phonemes present in the transcript.

3. Extract acoustic features from the audio

   Mel Frequency Cepstral Coefficients (MFCC) are the most commonly used features, but Perceptual Linear Prediction (PLP) features and other features are also an option. These features serve as the basis for the acoustic models.

Kaldi Training Overview

4. Train monophone models

A monophone model is an acoustic model that does not include any contextual information about the preceding or following phone. It is used as a building block for the triphone models, which do make use of contextual information.

5. Align audio with the acoustic models

Each training step will be followed by an alignment step where the audio and text can be realigned.

6. Train triphone models

A phonetic decision tree groups triphones into a smaller amount of acoustically distinct units, thereby reducing the number of parameters and making the problem simpler

7. Re-align audio with the acoustic models & re-train triphone models

Repeat steps 5 and 6 with additional triphone training algorithms for more refined models.

Training Algorithms

Delta+delta-delta training computes delta and double-delta features, or dynamic coefficients, to supplement the MFCC features.

LDA-MLLT stands for Linear Discriminant Analysis – Maximum Likelihood Linear Transform. The Linear Discriminant Analysis takes the feature vectors and builds HMM states, but with a reduced feature space for all data.

SAT stands for Speaker Adaptive Training and performs speaker and noise normalization by adapting to each specific speaker with a particular data transform.

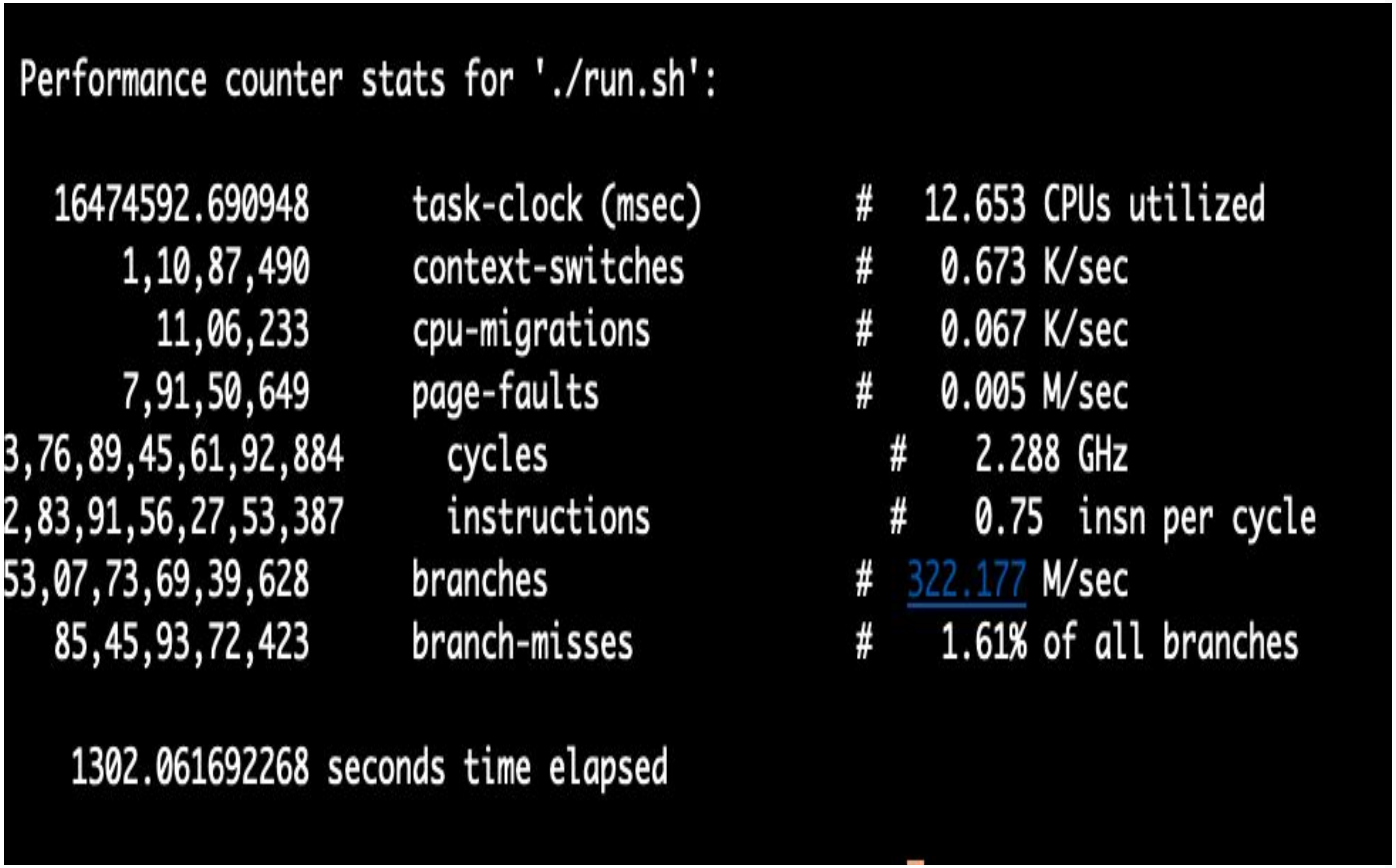
Alignment Algorithms

The actual alignment algorithm will always be the same; the different scripts accept different types of acoustic model input.

fMLLR stands for Feature Space Maximum Likelihood Linear Regression.

After SAT training, the acoustic model is no longer trained on the original features, but on speaker-normalized features. These quasi-speaker-independent acoustic models can then be used in the alignment process.

Kaldi Iban egs execution

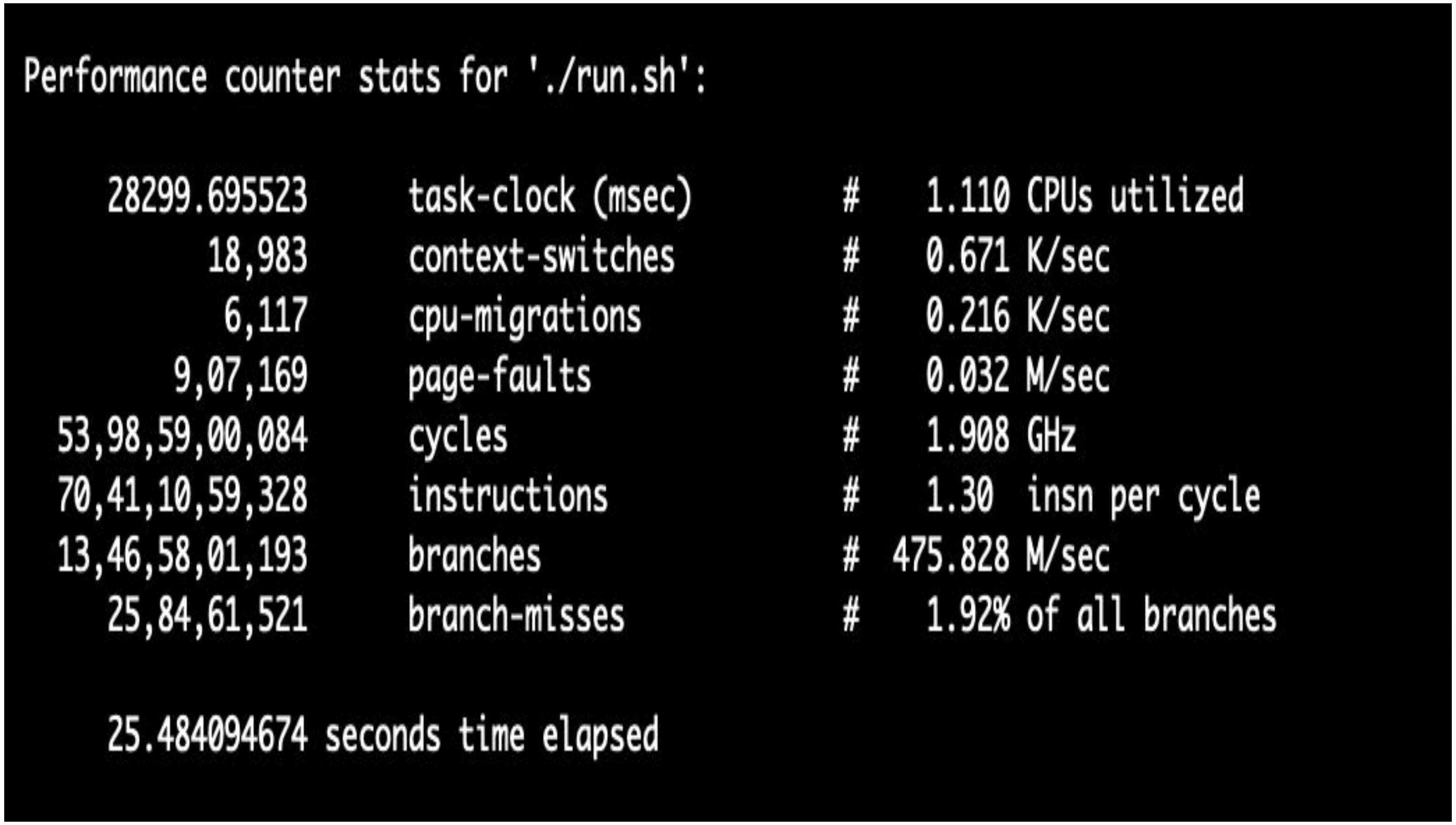


```
Performance counter stats for './run.sh':

   16474592.690948      task-clock (msec)         #   12.653 CPUs utilized
       1,10,87,490      context-switches          #    0.673 K/sec
         11,06,233      cpu-migrations            #    0.067 K/sec
       7,91,50,649      page-faults               #    0.005 M/sec
3,76,89,45,61,92,884      cycles                    #    2.288 GHz
2,83,91,56,27,53,387      instructions              #    0.75  insn per cycle
53,07,73,69,39,628      branches                  #  322.177 M/sec
   85,45,93,72,423      branch-misses             #    1.61% of all branches

    1302.061692268 seconds time elapsed
```

Kaldi Yesno egs execution



```
Performance counter stats for './run.sh':

     28299.695523      task-clock (msec)         #    1.110 CPUs utilized
           18,983      context-switches          #    0.671 K/sec
            6,117      cpu-migrations            #    0.216 K/sec
         9,07,169      page-faults               #    0.032 M/sec
  53,98,59,00,084      cycles                    #    1.908 GHz
  70,41,10,59,328      instructions              #    1.30  insn per cycle
  13,46,58,01,193      branches                  #  475.828 M/sec
     25,84,61,521      branch-misses             #    1.92% of all branches

     25.484094674 seconds time elapsed
```

Technologies Used

Operating System: Ubuntu 18.04
System Configuration : 2 x 8 core Intel Xeon 2.3Ghz, 64 GB DDR4 RAM, 1TB HDD

Languages used by the 8 applications in Tailbench and Kaldi:

● C
● C++
● Java
● Python and CUDA

Tools:

● Bash
● Perf
● Ftrace
● Valgrind
● taskset
● iostat
● dstat
● stress-ng
● cachestat
● htop
● matplotlib
● pandas
● scipy

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 response times for the requests increases on increasing the stress on the CPU and Memory.

● Caching is not a major factor for the performance of the application. The mean response time remains unchanged irrespective of the number of page faults.

● Although the mean response time for the requests increases, the ratio of the 95th and 99th percentile to the mean remains approximately the same.

Future Work - Tailbench/Sphinx

- 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 the foundation for analyzing Speech Recognition systems has been laid by profiling Sphinx, similar methods can be applied to profile Kaldi which is a better and faster ASR system written in C++ that uses Neural Network Models.
- The main challenge in analyzing Kaldi is that it requires high amounts of GPU memory.