Tell-Tale Tails: Decomposing Response Times for Live Internet Services

Nan Deng
The Ohio State University
dengn@cse.ohio-state.edu

Zichen Xu
The Ohio State University
xu.925@osu.edu

Christopher Stewart
The Ohio State University
cstewart@cse.ohio-state.edu

Xiaorui Wang
The Ohio State University
wang.3596@osu.edu

***Abstract*—Internet services use a wide range of software during request processing. One very slow software component can increase response time significantly. Or, many slightly slow components can conspire to increase response time. This paper 1) describes an approach to model the slowdown caused by each software component and 2) diagnoses root causes of tail response times in live services. Our approach uses Independent Component Analysis (ICA) to decompose the response time of independent, parallel requests into normalized software delays. Our approach models delays with less than 17% error for a wide range of software. It also captures the degree of data parallelism for each component, a measure of normalized energy footprint. We applied our approach to 33 services hosting real users, e.g., Google, Bing and Twitter. We observed that the services with slowest tail response times were affected by many components conspiring to slow down response time. We also found that energy footprint and tail response time were correlated.**

I. Introduction

Internet services try to respond to all requests quickly. A 2-second delay in response time decreases revenue by 17% [8]. However, for many services, the slowest 5% of requests (i.e., the tail) have much slower response time than normal requests. Among European e-commerce sites, 26% of requests respond more than 42% slower than the median response time [8]. At Microsoft Bing, the 99^{th} percentile of processing times can exceed median processing times by 5,600% [15].

Databases, key value stores, business logic or other software components can cause slow response time. On a case by case basis, caching, replication or other systems techniques can speed up slow software. However, the software that causes slow response time varies from service to service. System managers need to identify which software causes slow response time. Managers can directly measure delay caused by software under their control, but it is challenging to identify software components that slow down other services.

For this paper, we devised an approach to model the normalized delays caused by underlying software. We applied this approach to 33 real Internet services hosting live workloads and studied salient features of the recovered components in relation to tail response time.

Our first challenge was to model software delays for services hosting live workloads. Our approach is black box; it does not require modifying back-end software. Instead, we used independent component analysis (ICA) to decompose response time into per-component delays. ICA is a well known machine learning algorithm that extracts source signals from multiple, independent composite signals. In our context, source signals are the delays caused by software invocation during request processing. Composite signals are response times observed from independent and parallel requests. ICA assumes component delays are non-Gaussian. This assumption is reasonable because widely used software is known to have fat tails. ICA exploits fat tails to learn about delays caused by software (*i.e., tell-tale tails*).

After we decomposed response times into software delays, we defined statistical confidence thresholds to prune recovered components that were unstable. Then, we labeled components using a library that included widely used software, e.g., MySQL, Redis, MongoDB, etc. Our library also models the degree of data parallelism within request executions.

We validated our approach using benchmarks and real services hosting live workloads. We extracted component delays with 5% error on average and 17% error in the worst case. We used our approach to detect software used by each service. For our benchmarks, we accurately detected MySQL and PostgreSQL databases with 90% true-positive rate and less than 45% false-positive rate. For the real services, we used data in the public domain to confirm that 71% of our detected software components are actually used in practice. Looking into the salient features of the recovered components, we present key findings:

1. Services with slowest 95^{th} percentile response time likely suffer from request executions where many components exhibit slowdown. Perhaps surprisingly, services with fastest tail response time do not eliminate fat-tail components. Instead, these services mask the effects of slow components on response time.
2. Normalized software delays for MySQL, MongoDB and ElasticSearch can be detected across a wide range of environments.
3. The degree of data parallelism is positively correlated to tail response time (correlation coefficient is 0.908). As a proxy for normalized energy, this finding suggests that reducing energy footprint improves tail response time.

The remainder of this paper is as follows. Section II describes our approach to decompose response time into normalized, per-component delays. Our approach is a novel composition of ICA, heuristics to prune over fitting, and K-

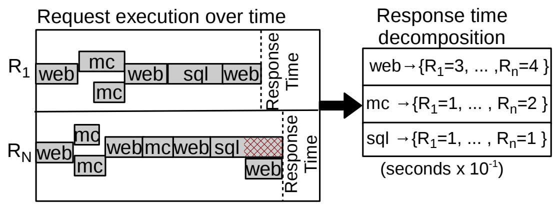


Fig. 1: Decomposing response times into delay caused by each software component.

nearest neighbor clustering. We validate our approach using benchmarks in Section III. Section IV presents key findings on per-component delays of real-world services. Section V describes related work and Section VI presents conclusions.

II. Methodology

Figure 1 shows the execution of PHP scripts (web), Memcached (mc), and MySQL (sql) during request processing. The web and sql components run in sequence using remote procedure semantics. The mc component uses parallel threads with each thread processing keys in its partition range. Finally, the web component triggers timeouts when the sql component is too slow. When this happens, the sql component finishes executing in the background while web completes the request.

Our goal is to decompose response time into the delays caused by each software component. Figure 1 depicts the response time decomposition. First, we must define software delay. In our context, software components comprise a code base that is repeatedly invoked during request processing. Software delay is the portion of response time spent executing a code base *and* waiting for CPU, memory and disk resources to execute it. In Figure 1, the sql component does not cause software delay after web times out, because background execution does not increase response time. Likewise, the slowest mc thread causes software delay but other, parallel mc threads do not.

Prior work time stamps the start and end of software invocations to decompose response time. For this paper, we used a black box approach shown in Figure 2. We did not change software or operating systems on the back-end servers. First, we measure response time for a target service by issuing many requests in parallel and over time. Then, we use ICA to recover the distribution of per-component software delay. We then prune our results for stability. Finally, we use a library to label recovered components.

Limitations: Our black-box approach is useful when Internet services block access to their back-end servers. However, it is not as powerful as direct instrumentation. By comparing Figure 1 and Figure 2, we highlight the following limitations: 1) Our approach returns *normalized* software delay, not actual delays. Here, normalized means that software delays are shifted to have zero mean and unit variance. As a result, we can not directly compare delays between two recovered components. 2) Our approach captures the distribution of software delay, not per-request delays. We can not explain slow response time for a specific request. 3) Our approach does not recover delay for every software component used during request processing. There may be other components that affect response time as well.

Despite these limitations, our approach helps managers identify components with fat tails, label hidden components, and model energy footprints.

A. System Model

We model software delay as a stochastic process with a linear multiplier. Random variable s_i is software delay per invocation, where i indexes components. Components are invoked a_i times during request processing. The total software delay for a request is $a_i \times s_i$. Considering a request invokes I components, then its response time x is a random variable which is a linear combination of all invoked components' delays, i.e. $x = \sum_{i=1,\ldots,I} a_i s_i$. In this paper, we use vector representation $x = \mathbf{a}^T\mathbf{s}$, where $\mathbf{a} = (a_1, \ldots, a_I)^T$ and $\mathbf{s} = (s_1, \ldots, s_I)^T$. Bold lowercase letters to represent vectors. Assuming per-component software delays comprise the majority of end-to-end response time, we let x_n be the response time of the n^{th} request and $\mathbf{x} = (x_1, \ldots, x_N)^T$ represents response times of N concurrent requests. Suppose the n^{th} request invokes the i^{th} component $a_{n,i}$ times, then we have:

$$\mathbf{x} = A\mathbf{s} \qquad (1)$$

The mixing matrix A is unknown. Our goal is to find software delays ($\mathbf{s}$) by only observing the response times ($\mathbf{x}$).

B. Extracting Per-Component Delays with ICA

To be sure, it is impossible to solve Eq (1) using only response times ($\mathbf{x}$) without constraining delays ($\mathbf{s}$) or the mixing matrix A. The number of unknowns exceeds the number of observations. A key contribution for this paper is the identification of practical constraints that 1) capture common operating conditions for Internet services and 2) allow us to solve Eq (1). The constraints are:

1. **Per-component delays are non-Gaussian:** It is well known that software delays in Internet services often have fat/heavy tails [22].
2. **Normalized component delays are independent:** Software delays depend on the processing speed of their underlying hardware. However, normalizing to zero mean and unit variance, makes these delays statistically independent. Software delays also depend on queuing. However, autoscaling [10] and multi-path networks significantly reduce queuing variance, limiting the affect of queuing delay on normalized delays.
3. **Invocation frequencies vary between requests:** Request parameters affect the invocation frequency of software like databases and Memcached.

Given the above constraints, ICA can be used to recover normalized software delays using only response times. ICA reverses Eq (1) by finding the mixing matrix W that is most likely $W = A^{-1}$. Specifically, ICA explores candidate W

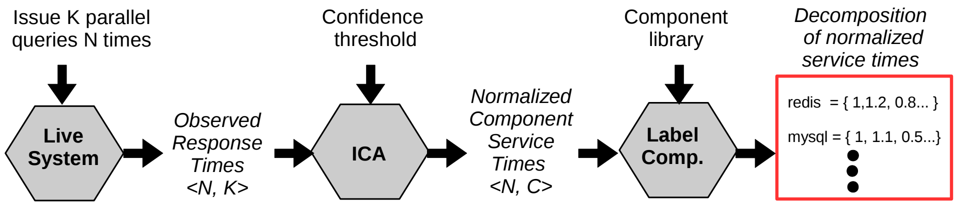

Issue K parallel queries N times
Live System
Observed Response Times <N, K>
Confidence threshold
ICA
Normalized Component Service Times <N, C>
Component library
Label Comp.
Decomposition of normalized service times
redis = { 1,1.2, 0.8... }
mysql = { 1, 1.1, 0.5...}

Fig. 2: Our approach to decompose response time into *normalized* software delays. It does not require changing or monitoring back-end servers.

matrices and chooses the matrix that minimizes mutual information between software components and Gaussianity within components. Minimizing mutual information maximizes independence of normalized delays [4], whereas minimizing Gaussianity constrains ICA to realistic mixing matrices. We use FastICA [14], one possible implementation of ICA that uses gradient descent methods to explore candidate W matrices. FastICA is a randomized, fixed-point, parameter-free algorithm whose convergence is cubic.

Choosing Request Parameters: Remember that the response times we collected are a set of vectors. These vectors are realizations of the random vector $\mathbf{x}$. Within each vector, there are response times from concurrent requests which invokes components with different frequency to make sure the rows in the mixing matrix A are linearly independent. Between those vectors, they should be the same set of requests to make sure that the mixing matrix A is the same across the experiment. Request parameters must be chosen carefully based on the presumed design of the target service.

C. Finding Stable Components

FastICA uses randomized, gradient descent. If it is executed twice on the same service, it may recover different component delays. This can happen for two reasons. First, the service can change in between the two executions. For example, CRON jobs that run only at night may shift software delays. Second, FastICA may converge upon a W matrix that is a local minima, introducing false-positive components.

We use two thresholds to build confidence that the recovered components reflect true software delays under normal operating conditions. The first threshold T_0 is a percentage, ensuring that recovered components are found in more than T_0 of FastICA executions. The second threshold T_1 sets the minimum similarity between components recovered across multiple executions. It is an absolute relative error. Recovered delays across two executions are from the same component if 1) their absolute relative error is less than T_1 and 2) for both components, there does not exist another recovered component with lower absolute error.

D. Labeling Components

Finally, our approach uses recovered delay distributions to infer the underlying code base. The key assumptions here are that 1) widely used software will have unique normalized delay distributions and 2) ICA can recover delays with sufficient accuracy to distinguish components. We use K-nearest neighbor clustering to match recovered delays to a library. Our library includes software components deployed with different levels of data parallelism. Thus, a match describes the recovered component's code base and energy footprint.

III. Validation

To validate the accuracy of our component delay recovery method, we compare delays for the recovered components to the observed software delays obtained by instrumenting the system. We setup a two-tier storage service. In each tier, there are several options of software to run. The first tier runs Memcached, Redis or ZooKeeper which are in-memory key-value storage software. The second tier runs MySQL, PostgreSQL, MongoDB or ElasticSearch. This gives us 12 possible configurations for the system

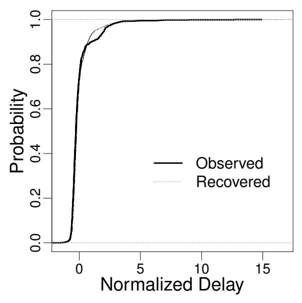
We run our experiments on virtual machine instances from Google Cloud Compute Engine. For MySQL, PostgreSQL, Memcached, MongoDB and Redis, we run them on single "n1-standard-1" instances, which has one CPU core from 2.5GHz Intel Xeon E5 v2 (Ivy Bridge) and 3.75GB of memory. ZooKeeper runs in a 3-node cluster where each node is an "n1-standard-1" instance. ElasticSearch runs on a two-node "n1-standard-1" cluster. All experiments are conducted within "us-central1-f" region, which is located in Council Bluffs, Iowa.

The system accepts 6 types of requests, where each of them triggers a write operation in each tier with different frequency. In each experiment, we issue these 6 requests types concurrently and repeatedly send such concurrent requests every 500ms for 1000 times. This results 1000 sets of response times where each set contains response times from 6 concurrent requests with different types. For each configuration, we repeat this experiment for 100 times. We set $T_0 = 50\%$ and $T_1 = 3\%$ to prunes false components.

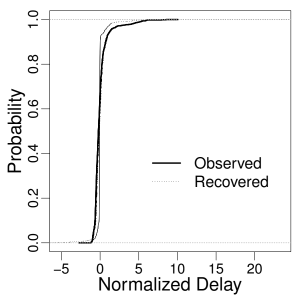
A. Accuracy of Recovered Component Delays

The first question we want to answer is that if ICA could accurately recover software delays. We know that ICA could recover statistically mutually independent component delays. According to the discussion in previous section, if software delays are independent, then ICA could in theory recover them.

Comparing one recovered component with a set of software delay requires a metric to measure the distance of two sets of 1000 samples of delays. We use symmetric Kullback-Leibler divergence [3], or KL divergence, as our metric to measure the distance. Intuitively, given two sets of samples, KL divergence tells how many additional bits are required to represent one



(a) Observed and recovered delays for a component that yields median error (error=0.5%).



(b) Observed and recovered delays for a component that yields 90^{th} %tile error (i.e., near worst case,error=15.6%).

Fig. 3: Cumulative distribution functions for software delays. Delays are normalized to zero mean unit variance.

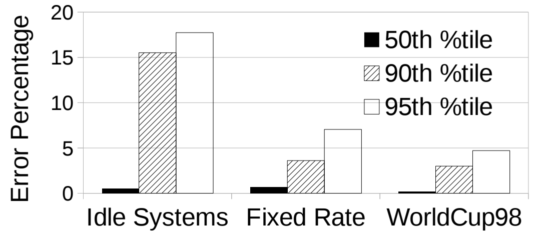


Fig. 4: 50th, 90th and 95th percentile errors of the recovered component delays under different workloads.

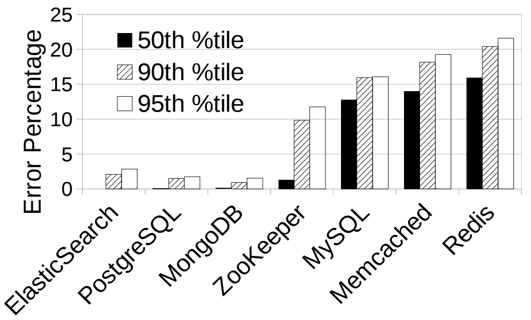


Fig. 5: 50th, 90th and 95th percentile errors for different software. There is no background workload running on the test systems.

set of samples with another. The maximum divergence for 1000 samples is 17.667 bits. Relative error (reported as a percentage) is KL divergence divided by this number.

Given an experiment, every recovered component's delays is matched with the closest software delays monitored from the experiment. We then collect KL divergence of those matches in all experiments under all configuration. We find that the recovered component delays could be used to accurately approximate software delays. The 50th and 90th percentiles of the error between component and software delays in all experiments are 0.5% and 15.5%. Figure 3 shows the corresponding empirical cumulative distribution functions (eCDFs). As we discussed in Section II, the component delays that ICA recovers are normalized to zero mean unit variance. The X-axis in these eCDFs is normalized delays. Remember that our result considers all experiments (100 experiments for each configuration) under all configurations (3 tier-1 software and 4 tier-2 software). As shown in these figures, ICA could accurately recover software delays in almost all cases.

B. Recovered Delay Accuracy under Different Workload

In the previous subsection, our experiment is conducted under idle systems. There is no outside workload other than our probing requests. We would also want to know how well ICA recovers delays under live workloads.

We set up the same two-tier storage service again and probed the system at the same rate. This time, we test our service under two workloads. The first workload is fixed-rate sending 300 requests per second which contains 100 read requests and 200 write requests. The second is the WorldCup98 [1] workload with time varying workload.

Figure 4 shows the results of the median, 90th and 95th percentile errors of recovered delays under different workloads. We can see that adding workload in the background does not hurt the accuracy. In fact, it even improves the accuracy in the worst cases. Remember that ICA recovers component delays that are statistically mutually independent and non-Gaussian. Applying workloads skews the component delay distributions in each tier, making them less like a Gaussian distribution. Note, this test compares recovered component delays to observed delays under the same workload. Later, we will study the effect of changing the workload, i.e., comparing recovered delays under one workload to observed delays under a different workload.

C. Recovered Delay Accuracy for Different Software

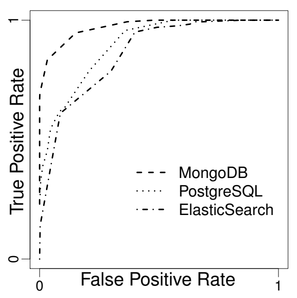
The next question we want to answer is that if the accuracy of recovered delays differs from software by software. The answer turns out to be positive. Some software components are recovered with lower error. Figure 5 shows 50th, 90th and 95th percentiles of KL divergence between software and component delays grouped by software. Note that ElasticSearch, PostgreSQL and MongoDB's delays could always be recovered accurately; while Memcached and Redis' recovered component delays are bit far from their software delays. It is not surprised because fast in-memory key-value stores like Memcached and Redis have relatively small delays making it easily to be masked by other components' delays.

D. Identifying Software by Recovered Component Delays

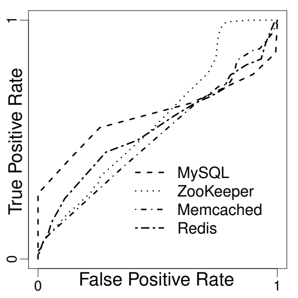
To push our study even further, we want to see if we could use recovered component delays to identify the underlying software running inside a service. That is, can we extract components accurately enough to distinguish their normalized distribution from other software.

We built a library of normalized software delays measured under idle workload for Memcached, Redis, ZooKeeper, MySQL, PostgreSQL, MongoDB and ElasticSearch. We collected 10 sets of $10,000$ delays for each software component.

To identify the underlying software in the system, we build one binary classifier for each software. Each classifier takes a



(a) ROC curves of classifiers with low errors



(b) ROC curves of classifiers with high errors

Fig. 6: Receiver operating characteristic (ROC) curves of classifiers for each software component when K=3.

set of recovered component delays as input and returns positive if the set of delays is close enough to the corresponding software's delay. The algorithm of our classifier is quite simple and could be summarized into one sentence: *Return positive if in the library there are more than K sets of software delay samples whose KL divergence with the input component delay is less than $KL_{threshold}$ bits.* Two parameters are K and $KL_{threshold}$. Since in our library, each software has 10 sets of delays, K is between 1 and 10. As we mentioned before, the recovered component delays could not be equally accurate for all software, $KL_{threshold}$ differs from software to software. Because we build one classifier per software, there are 7 classifiers in total in our test.

To measure the performance of the classifiers, we use receiver operating characteristic (ROC) curves to show the results. ROC curve is a widely used graphical tool to illustrate the performance of a binary classifier. The basic idea of the ROC curve is to run the classifier under different parameters on the same data set and record its false positive rate and true positive rate. The Y-axis of an ROC space is the true positive rate, which is the division of number of true positives (in our case, it means the system contains the software and the classifier could correctly identifies it) over the number of positives (in our case, it means the system contains the software) in the data set. The X-axis is the false positive rate, which is the division of the number of false positives (in our case, it means the system does not have the software but the classifier falsely identified the software) over the number of positives. Both true and false positive rates are between zero and one. To draw an ROC curve, we pick a fixed value for K, change $KL_{threshold}$ to get different values of true-positive and false-positive rate and draw those points in the ROC curve.

Figure 6 shows ROC curves when $K = 3$. The diagonal in the ROC space means a classifier that performs random guess. Any point above the diagonal in the ROC curve means a possible configuration which is better than a random guess. We can clearly see that the performance of classifiers is highly correlated with the accuracy of the recovered component delays. We group the ROC curves in two figures: Figure 6(a) shows the software whose delays could be recovered with low error. These software can be accurately identified by their

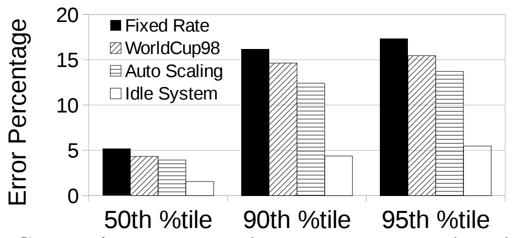


Fig. 7: Comparing recovered components under different workload against a library. The library is built using software delays in an idle system.

classifiers. In the case of MongoDB, the true positive rate could reach 98% while the false positive rate could be still under 20%. On the other hand, software, whose recovered delays has medium error, like MySQL and ZooKeeper could be identified but sacrifices true positive rate for lower false positive rate.

However, software delay distributions will be skewed by its workload. This restricted our approach because the workload of software is unknown when the library is built. Fortunately, modern cloud computing principles mitigates this problem. Principles, especially like auto scaling, make per server workload stable by dynamically adding or removing resources to each tier when the outside workload changes. We conduct an experiment on the storage system running Memcached and MongoDB in each tier serving WorldCup98 day 70's workload. Each tier scales individually along with the changes of the workload. We then compare the recovered component delays against a library built with Memcached and MongoDB delays in an idle system. As a comparison, we also run fixed-rate (300 rps) and WorldCup98 workload on another system without auto-scaling. Figure 7 shows the median, 90th and 95th percentile error by comparing recovered component delays with software delays in our library. We can see that auto-scaling decreases the difference.

E. Exploring Parallelism within Components

As we described in Section II, in our model, a request is processed sequentially through independent components. There is no parallelism taken into consideration when we apply ICA on response times.

Since we have shown above that with a help of a library of known software, we could match the recovered components' delay back to software within the library. We would like to further use the library to explore the parallelism within a component.

Looking into one tier in the system, consider a request is being processed in parallel within a cluster of servers. Assuming that the request can only be processed by the next tier only if N servers respond the request. We call the number N as the *parallel factor* for the software in the tier. The delay of the tier is the maximum delay of the N servers, which would be recovered by ICA.

To detect the parallel factor of a given software, we adopt the same idea of using a library of known software. We could

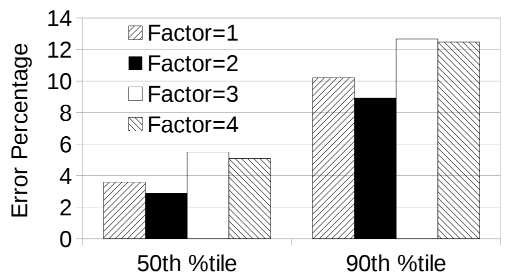


Fig. 8: 50th and 90th percentiles of KL divergence between component delays and re-sampled MongoDB delays with parallel factors of 1, 2, 3 and 4. The actual parallel factor in the system is 2.

setup the software on one server under a controlled environment and collects its delays. We only keep the maximum delay of every N delays and build a new set of delays. If a recovered component's delay is close to the set of delays, we would say that the recovered component has a parallel factor of N.

To validate our approach, we setup the two-tier storage service, and put a 2-node MongoDB cluster in the second tier. For the first tier, we use one node to run Memcached. We re-use the same MongoDB delays from the training set used in Section III-D but uniformly re-sample them (with replacement) by keeping the maximum of every 2, 3, and 4 delays and construct 10 sets of delays for each parallel factor. Again, in each set, there are 1000 samples of delay.

We then apply ICA and recover independent component delays from the system. Each recovered component delay is compared with the re-sampled MongoDB delays within the training sets using symmetric KL divergence. Figure 8 shows the 50th and 90th percentiles of KL divergence between the recovered component delays and re-sampled MongoDB delays with parallel factors of 1 (i.e. the original MongoDB delays), 2, 3, and 4. As we can see, the re-sampled delays with parallel factor of 2 is the closest to the recovered component delays. This agrees the fact that we use a 2-node MongoDB cluster in the backend of the system.

IV. Study on Real Internet Services

We applied our approach to decompose response time for 33 real Internet services that support keyword search. Requests that execute keyword search meet the requirements outlined in Section II:

1. **Keyword search uses software with non-Gaussian service times:** ElasticSearch, Apache Solr, Memcached, Redis and SQL databases are commonly used to process keyword searches. As shown in Section III, these components have fat tail, non-Gaussian execution times.
2. **Keyword searches can have independent normalized delay:** Our search parameters include long words composed from random letters. These words subvert caches that would make normalized delay between requests interdependent. Each of our services are likely to proceed through all layered caches before returning a result.
3. **Invocation frequencies vary:** Our requests also use search parameters that specify real keywords and categories. Under live workloads, these parameters ensure concurrent requests will invoke cache components differently.

We selected a wide range of services from large popular sites like Google and Amazon to smaller sites like Sundial (a comedy magazine). All services support HTTP/HTTPS, allowing us to use CGI to specify keyword parameters. For each service, we issued 5 concurrent requests 1000 times, allowing 500ms idle time between each round. We conducted 20 experiments for each services, spreading experiments over 30 days. We set T_0 to 50%, meaning we pruned components that appeared in fewer than 10 experiments. We set T_1 to 3% error, meaning components were considered to represent the same software component if the absolute relative error between their distributions was less than 3%.

A. Component Delays and Response Times

Figure 9 compares tail response time and component delay for each service. Slow components do not necessarily cause slow response time. Craiglist, Youtube, Yelp, Google and Amazon use components with relatively fat tails but achieve skinny tails for response time. The 11 services that achieve fastest response-time tails support at least one component with a longer tail. To be sure, Figure 9 reports normalized delay. We can not directly compare tails for response time and components. However, it is likely that these services are engineered to survive slow responding components. Either slow components make up a small portion of response time or the services react when components take too long (e.g., timeouts).

Figure 9 also shows that services with poor tail response time are affected by multiple components. The 5 of the 6 services with slowest tail response time perform worse than their slowest component. Response time tail results from multiple components executing slowly at the same time.

B. Labeling Software Used in Real Sites

We have shown in the previous section that it is possible to use recovered component delays and a library of known software to roughly find what software is running in the backend system given an Internet service's response times. Besides, we could further manipulate the library to find how many instances of a given software is used in parallel to process a user request. Number of parallel instances involved in each request could be used as a rough estimation of the energy footprint per request of the Internet service.

We apply the same technique on real Internet services' data and try to find what software runs behind those services. We also consider different parallel factors for each software. For each software and a given parallel factor, we build a binary classifier comparing the recovered delay with the software delays in the library. The classifiers are tuned using our two-tier storage service's data. Their parameter K is set to maximize the area under their ROC curve (AUC); and $KL_{threshold}$ is

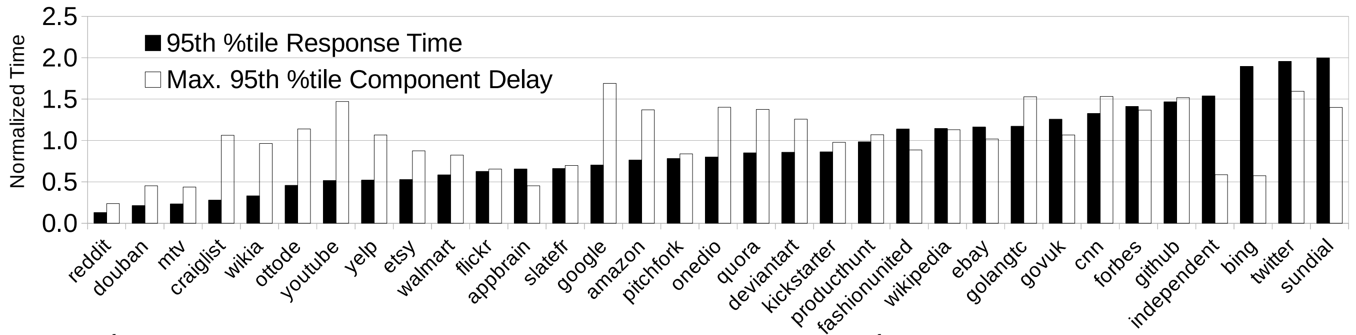


Fig. 9: 95^{th} percentile of normalized response times compared to the largest 95^{th} percentile of recovered software delays for 33 real Internet services.

	Found & Confirmed	Found & Unconfirmed
ElasticSearch	Ebay, Etsy, Kickstarter, Walmart, Yelp	Bing, Deviantart, Slate.fr
MongoDB	FashionUnited, Gov.uk, Mtv.com, Otto.de, Slate.fr	Ebay
PostgreSQL	AppBrain	Yelp, Walmart
MySQL	Ebay, Flickr, Github, Kick-Starter, Pitchfork, Twitter , Walmart, Wikia	CNN.com , Etsy, The Independent
ZooKeeper	Reddit	Etsy, Flickr, KickStarter

TABLE I: Results of finding software running in Internet services. *Found & Confirmed* means the software is found by our classifier and we can find at least one reliable source confirming that the service uses the software. The numbers in parenthesis after each service's name are parallel factors.

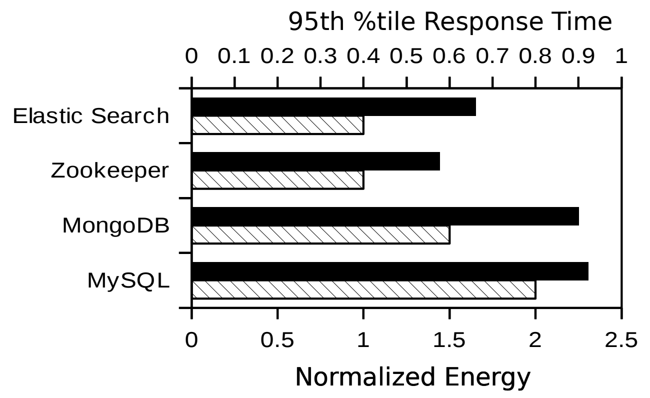


Fig. 10: 95th percentile response times and normalized energy footprint. The black bars are the median of the 95th percentile response time of the sites that are found using the software by our classifier. The shaded bars are the median normalized energy footprints (or parallel factors) of those sites.

chosen to maximize the true positive rate while maintains the false positive rate less than 1/3 of the true positive rate.

Table I shows the software that we found for each Internet service. For recovered component that matched software components in our library, we searched for public-domain data that confirmed that the service actually uses the matched software component. Specifically, we searched technical blogs, official Powerpoint slides, white papers and reliable third party articles. Even though different services are running different versions of software under diverse environments, our classifiers could still have a decent performance partly because we use normalized software delays. In most cases, once our classifier finds a software running behind a service, it could be confirmed by at least one source. However, there are some obvious errors that we can see from the table: Microsoft Bing is very unlikely to use ElasticSearch (though, they may use similar proprietary software). Yelp and Walmart use MySQL, making it unlikely that they also use PostgreSQL.

We matched 28 recovered, stable components against the open source components in our library. We were able to confirm 20 of the components were used in production by the services (71% success rate).

C. Studying Normalized Energy Footprint of Real Sites

Recall, our matching approach discovers the parallel factor for each component in library, i.e., an estimate on the number of data-parallel software invocations during request execution. Assuming that data-parallel optimizations linearly increase energy footprint, the parallel factor describes normalized energy footprint.

In Figure 10, we noticed that increasing data parallelism would increased the tail response time. We grouped energy footprints and 95th percentile response time by the software component matched in our library, reporting the median for both energy footprint and response time. As we can see in the figure, higher tail response time is highly correlated to higher energy footprints. The correlation coefficient of them is 0.908.

V. Related Work

Prior work traces request execution across components by changing their source code, or their running environment, e.g. using a modified kernel. The Dependable Compute Cloud framework measures CPU and network usage and uses machine learning techniques to infer per-component needs [11]. Magpie [2] changed the operating system to account for component needs. X-Trace [9] and Ubora [16] account for component needs by tracking network communications with some stronger assumptions about service design.

Per-component resource accounting improves anomaly detection. PerfScope [5] analyzes recent system calls to perform online bug inference. It narrows down the possible buggy functions by detecting time or frequency changes in system calls. PREPARE [23] monitors virtual machines' system-level metrics and applies statistical learning algorithms to detect performance anomalies. It works at the hypervisor level making it possible to be applied by cloud providers. PowerTracer [19] and Power Containers [20], [21] maps the low-level measurements back to request context. These low-level traces were combined to produce diverse views of the system ranging from per-node system call counts to per-tier energy efficiency. Li et al. [18] studied the relationship between these low-level metrics and the tail latencies of components in Internet services. Their findings shown that scheduling policy, CPU power saving mechanisms and NUMA effects would significantly affects the tail latencies.

Software log messages are also useful to conduct a component-level study of an Internet service. Xu et al. [24] use static analysis to find log statements in the source code and relate the anomalous log messages back to the source code. When these messages are too numerous, they can be subsampled to reduce overhead for lower quality [12]. However, log messages can only be retrieved with access to the back-end system.

Our non-invasive approach allows third-party companies to infer normalized per-component needs. Green hosting is an approach to improve the sustainability of Internet services by offsetting the carbon footprint of target components [7]. More generally, many sustainable computing systems attempt to target specific components and users [6], [13], [17], [25]. An ICA-based approach can distinguish such users without impractical and extensive source code changes.

VI. Conclusion

This paper presents a series of study on decomposing response times into per-component delays. Using Independent Component Analysis (ICA), response times can be accurately divided into component delays with 90th percentile error less than 17%. Using a library of known software, these recovered component delays can be used to find the software running behind an Internet service and its corresponding parallel factor, which is directly related to per-request energy footprint. Further, 33 real Internet services' per-component delays are studied along with their response times. We find that the collaboration between components in a service is important to the system's performance. Also, we find that the energy footprint and tail response time are correlated.

VII. Acknowledgments

Our work was supported by NSF grants IIS-1117699, CNS-1143607 (NSF CAREER Award), and CNS-1350941 (NSF CAREER Award). We thank the reviewers, Aniket Chakrabarti, Chen Li, and Alex Bunch for early feedback.

References

[1] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3), 2000.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[3] S. Boltz, É. Debreuve, and M. Barlaud. knn-based high-dimensional kullback-leibler distance for tracking. In *Eighth International Workshop on Image Analysis for Multimedia Interactive Services, WIAMIS 2007, Santorini, Greece, June 6-8, 2007*, page 16, 2007.

[4] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991.

[5] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 8:1–8:13, New York, NY, USA, 2014. ACM.

[6] N. Deng, C. Stewart, D. Gmach, and M. Arlitt. Policy and mechanism for carbon-aware cloud applications. In *NOMS*, Apr. 2012.

[7] N. Deng, C. Stewart, J. Kelley, D. Gmach, and M. Arlitt. Adaptive green hosting. In *International Conference on Autonomic Computing*, 2012.

[8] T. Everts. Case study: Understanding the impact of slow load times on shopping cart abandonment. http://blog.radware.com/, 2013.

[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007.

[10] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf. Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward. In *ACM SIGMETRICS*, 2013.

[11] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *International Conference on Autonomic Computing*, 2014.

[12] I. Goiri, R. Bianchini, S. Nagarakatte, and T. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM ASPLOS*, 2015.

[13] M. E. Haque, K. Le, Í. Goiri, R. Bianchini, and T. D. Nguyen. Providing green slas in high performance computing clouds. In *International Green Computing Conference (IGCC)*, pages 1–11. IEEE, 2013.

[14] A. Hyvrinen and E. Oja. A fast fixed-point algorithm for independent component analysis. *Neural Computation*, 9:1483–1492, 1997.

[15] M. Jeon, Y. He, S. Elnikety, A. Cox, and S. Rixner.

[16] J. Kelley, C. Stewart, D. Tiwari, Y. He, S. Elnikety, and N. Morris. Measuring and managing answer quality for online data-intensive services. In *International Conference on Autonomic Computing*, June 2015.

[17] C. Li, R. Wang, T. Li, D. Qian, and J. Yuan. Managing green datacenters powered by hybrid renewable energy systems. In *International Conference on Autonomic Computing (ICAC)*, 2014.

[18] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[19] G. Lu, J. Zhan, H. Wang, L. Yuan, and C. Weng. Powertracer: Tracing requests in multi-tier services to diagnose energy inefficiency. In *Proceedings of the 9th international conference on Autonomic computing*, pages 97–102. ACM, 2012.

[20] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM ASPLOS*, 2012.

[21] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *ACM ASPLOS*, Mar. 2008.

[22] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *International Conference on Autonomic Computing*, 2013.

[23] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS'12*, pages 285–294, 2012.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[25] Z. Xu, N. Deng, C. Stewart, and X. Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 177–186. IEEE, 2015.