Speeding up Distributed Request-Response Workflows

Virajith Jalaparti (UIUC) Peter Bodik Srikanth Kandula
Ishai Menache Mikhail Rybalkin (Steklov Math Inst.) Chenyu Yan

Microsoft

Abstract– We found that interactive services at Bing have highly variable datacenter-side processing latencies because their processing consists of many sequential stages, parallelization across 10s-1000s of servers and aggregation of responses across the network. To improve the tail latency of such services, we use a few building blocks: reissuing laggards elsewhere in the cluster, new policies to return incomplete results and speeding up laggards by giving them more resources. Combining these building blocks to reduce the overall latency is non-trivial because for the same amount of resource (e.g., number of reissues), different stages improve their latency by different amounts. We present Kwiken, a framework that takes an end-to-end view of latency improvements and costs. It decomposes the problem of minimizing latency over a general processing DAG into a manageable optimization over individual stages. Through simulations with production traces, we show sizable gains; the 99th percentile of latency improves by over 50% when just 0.1% of the responses are allowed to have partial results and by over 40% for 25% of the services when just 5% extra resources are used for reissues.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications*

Keywords

Interactive services; Tail latency; Optimization; Reissues; Partial results

1. INTRODUCTION

Modern interactive services are built from many disjoint parts and hence, are best represented as directed acyclic graphs. Nodes in the graph correspond to a specific functionality that may involve one or more servers or switches. Edges represent input-output dependencies. For example, Fig. 1 shows a simplified DAG corresponding to the web-search service at Bing, one of the major search engines today. In this paper, we use the term *workflow* to refer to such a DAG and *stage* to refer to a node in the DAG.

Analyzing production traces from hundreds of user-facing services at Bing reveals that the end-to-end response latency is quite variable. Despite significant developer effort, we found over 30% of the examined services have 95th (and 99th) percentile of latency 3X (and 5X) their median latency.

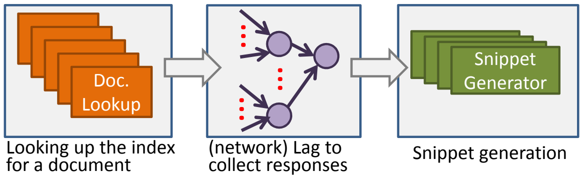

Doc. Lookup
Looking up the index for a document
(network) Lag to collect responses
Snippet Generator
Snippet generation

Figure 1: A simplified version of the workflow used for web-search at Bing.

Delivering low and predictable latency is valuable: several studies show that slow and unpredictable responses degrade user experience and hence lead to lower revenue [8, 9, 22]. Further, these services represent sizable investments in terms of cluster hardware and software, so any improvements would be a competitive advantage.

We believe that the increase in variability is because modern datacenter services have workflows that are long and highly parallel. In contrast, a typical web service workflow has a length of two (a web-server and a database) and a width of one. As we show in §2, the median workflow in production at Bing has 15 stages and 10% of the stages process the query in parallel on 1000s of servers. Significant delays at any of these servers manifest as end-to-end delays. To see why, as a rule of thumb, the 99th percentile of an n-way parallel stage depends on the 99.99th percentile of the individual server latencies for $n = 100$ (or 99.999th for $n = 1000$).

While standard techniques exist to reduce the latency tail [10], applying them to reduce end-to-end latency is difficult for various reasons. First, different stages benefit differently from different techniques. For example, request reissues work best for stages with low mean and high variance of latency. Second, end-to-end effects of local actions depend on topology of the workflow; reducing latency of stages usually off the critical path does not improve end-to-end latency. Finally, many techniques have overhead, such as increased resource usage when reissuing a request. For these reasons, latency reduction techniques today are applied at the level of individual stages, without clear understanding of their total cost and the achieved latency reduction. Therefore, without an end-to-end approach, the gains achieved by such techniques are limited.

In this paper, we present a holistic framework that considers the latency distribution in each stage, the cost of applying individual techniques and the workflow structure to determine how to use each technique in each stage to minimize end-to-end latency. To appreciate the challenge, consider splitting the reissue budget between two stages, 1 and 2, in a serial workflow. Fig. 2a shows how the variance of the latency of these two stages (Var1 and Var2, respectively) varies with the fraction of the total budget allocated to Stage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
Copyright © 2013 ACM 978-1-4503-2056-6/13/08…$15.00.

1 (on x-axis). Since both stages have similar variance when receiving zero budget, one may expect to divide the budget evenly. However, Stage 1's variance decreases quickly with reissue budget and the marginal improvement with additional budget is small. As marked in the figure, assigning the budget roughly 1:3 among the stages leads to the smallest variance of end-to-end latency (Sum Var). Comparing Sum Var with the 99th percentile latency in Fig. 2b shows that the sum of variances of the stages is well correlated with the 99th percentile, a fact that we will prove and use extensively.

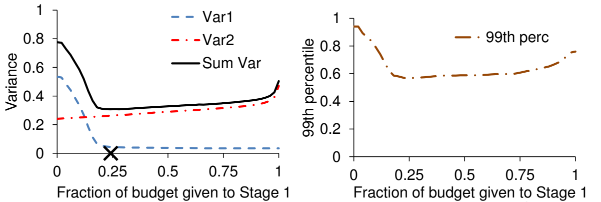
Kwiken formulates the overall latency reduction problem as a *layered* optimization relying on the fact that query latencies across stages are only minimally correlated. The first layer consists of *per-stage variance models* that estimate how the latency variance in individual stages changes as a function of budget allocated to that stage; these models may also incorporate other intra-stage optimizations where there are different ways of using budget within a stage. The workflow layer integrates the per-stage models into a single *global objective function* designed such that its minimization is well-correlated to minimizing higher percentiles of the end-to-end latency. The objective function also has a simple separable structure that allows us to develop efficient gradient-like methods for its minimization.

Further, we present two new latency reduction techniques: a new timeout policy tp trade off partial answers for latency and catching-up for laggard queries. The basic ideas behind these strategies are quite simple. First, many workflows can still provide a useful end-to-end answer even when individual stages return partial answers. So, at stages that are many-way parallel, Kwiken provides an early termination method that improves query latency given a constraint on the amount of acceptable loss on answer quality. Second, Kwiken preferentially treats laggard queries at later stages in their workflow, either by giving them a higher service rate (more threads), being more aggressive about reissuing them or by giving them access to a higher priority queue in network switches. Kwiken incorporates these techniques into the optimization framework to minimize the end-to-end latency while keeping the total additional cost low.

While in this paper we apply Kwiken in the context of request reissues and partial execution inside Bing, our solution applies more generally, for example to the network latency reduction techniques described in [23]. It also applies to most applications where work is distributed among disjoint components and dependencies can be structured as a DAG. This includes modern web services (e.g., Facebook [20] or Google [10]) and page loading in web browsers [24] and mobile phone applications [21].

We evaluate our framework with 45 production workflows at Bing. By appropriately apportioning reissue budget, Kwiken improves the 99th percentile of latency by an average of 29% with just 5% extra resources. This is over half the gains possible from reissuing every request (budget=100%). At stages that are many-way parallel, we show that Kwiken can improve the 99th percentile latency by about 50% when partial answers are allowed for just 0.1% of the queries. We, further, show that reissues and partial answers provide complementary benefits; allowing partial answers for 0.1% queries lets a reissue budget of 1% provide more gains than could be achieved by increasing the reissue budget to 10%. We also demonstrate robustness of parameter choices.

In summary, we make the following contributions:



(a) Variance (b) End-to-end 99th percentile

Figure 2: Impact of splitting the budget between two stages in a serial workflow on the variance of individual stages (Var1 and Var2), and the variance (Sum Var) and 99th percentile of the end-to-end latency (metrics are normalized)

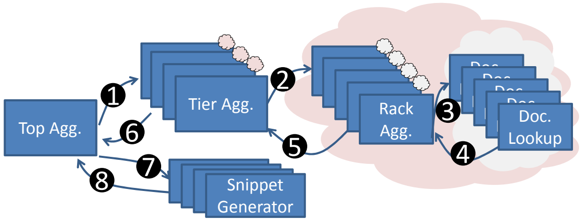


Figure 3: Timeline diagram of the processing involved for the workflow in Fig. 1.

- **Workflow characterization.** We describe low-latency execution workflows at a large search engine, analyze in detail the structure of the workflow DAGs and report on the causes for high variability.
- **New strategies.** We provide novel policies for bounding quality loss incurred due to partial answers and for catching-up on laggards.
- **Optimization framework.** We present a holistic optimization framework that casts each stage as a variance-response curve to apportion overall budget appropriately across stages. We evaluate the framework on real-world workflows and demonstrate significant reductions in their end-to-end latencies, especially in the higher percentiles i.e., tail latencies.

2. WORKFLOWS IN PRODUCTION

We analyze workflows from production at Bing to understand their structural and behavioral characteristics and to identify causes for slow responses.

2.1 Background

The workflow of an end-to-end service is a collection of stages with input-output dependencies; for example, responding to a user search on Bing involves accessing a spell checker stage and then in parallel, a web-search stage that looks up documents in an index and similar video- and image-search stages. Architecting datacenter services in this way allows easy reuse of common functionality encapsulated in stages, akin to the layering argument in the network stack.

Workflows can be hierarchical; i.e., complex stages may internally be architected as workflows themselves. For example, the web-search stage at Bing consists of multiple tiers which correspond to indexes of different sizes and freshness. Each tier has a document-lookup stage consisting of tens of thousands of servers that each return the best document for the phrase among their sliver of the index. These documents are aggregated at rack and at tier level and the most relevant

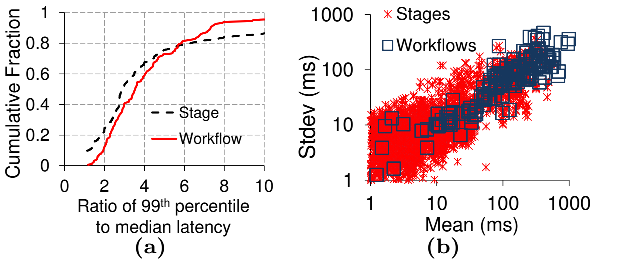


Figure 4: Estimating the variability in latency: (a) CDF of 99$^{\text{th}}$ percentile to median latency and (b) mean vs. standard deviation of latency, for stages and workflows.

results are passed along. This stage is followed by a snippet generation stage that extracts a two sentence snippet for each of the documents that make it to the final answer. Fig. 3 shows a timelapse of the processing involved in this workflow; every search at Bing passes through this workflow. While this is one of the most complex workflows at Bing, it is still represented as a single stage at the highest level workflow.

The observed causes for high and variable latency include slow servers, network anomalies, complex queries, congestion due to improper load balance or unpredictable events, and software artifacts such as buffering. The sheer number of components involved ensures that each request has a non-trivial likelihood of encountering an anomaly.

2.2 Workflow characteristics

We characterize most of the workflows and their latencies at Bing. We use request latencies from 64 distinct workflows over a period of 30 days during Dec 2012. We only report results for workflows and stages that were accessed at least 100 times each day, the 25th and 75th percentile number of requests per stage per day are 635 and 71428 respectively. In all, we report results from thousands of stages and hundreds of thousands of servers.

2.2.1 *Properties of latency distributions*

Latencies of stages and workflows have long tail. To understand variation of latency in workflows and in individual stages, Fig. 4a plots a CDF of the ratio of the latency of the 99$^{\text{th}}$ percentile request to that of the median request across the stages and workflows in our dataset. We see that stages have high latency variability; roughly 10% have 99$^{\text{th}}$ percentile 10X larger than their median. When the stages are composed into workflows, the variability increases on the low end because more workflows can have high variability stages but decreases the variability on the high end. Fig. 4b shows on a log scale the mean latency in a stage and workflow compared to the standard deviation. We see that the larger the mean latency in a stage the larger is the variability (standard deviation). However, stages with similar mean latency still have substantial differences in variability.

Stages benefit differently from reissues. Fig. 5 illustrates how reissuing requests impacts the latency for a subset of the stages from production. It shows the normalized variance in latency for these stages when a particular fraction of the slowest queries are reissued. Clearly, more reissues lead to lower variance. However, notice that stages respond to reissues differently. In some stages, 10% reissues significantly reduce variance, whereas in other stages even 50% reissues do not achieve similar gains. This is because the reduction in

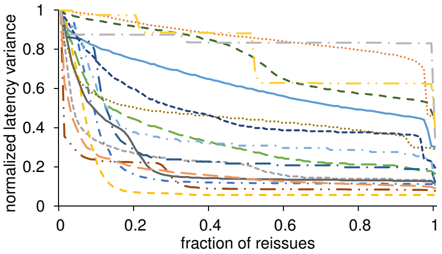


Figure 5: For a subset of stages in production, this plot shows normalized latency variance of the stage as a function of fraction of requests that are reissued.

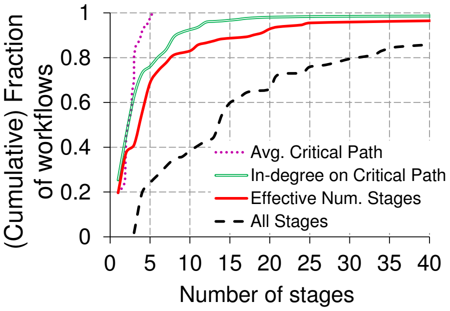


Figure 6: A few characteristics of the analyzed workflows

variance at a stage depends on its latency distribution: stages with low mean and high variance benefit a lot from reissues but the benefits decrease as the mean increases. Hence, giving every stage the same fraction of reissues may not be a good strategy to reduce latency of the workflow.

Latencies in individual stages are uncorrelated. We ran a benchmark against the most frequent workflow, where we executed two concurrent requests with same parameters and specified they should not use any cached results. These requests executed the same set of stages with identical input parameters and thus allowed us to study correlation of latencies in individual stages. We used 100 different input parameters and executed a total of 10000 request pairs. For each of the 380 stages in this workflow, we compute the Pearson correlation coefficient (PCC). About 90% of the stages have PCC below 0.1 and only 1% of stages have PCC above 0.5. Hence, we treat the latency of the first request and of the reissue as independent random variables.

Latencies across stages are mostly uncorrelated. To understand correlation of latencies across stages, we compute the PCC of latencies of all stage pairs in one of the major workflows with tens of thousands of stage pairs. We find that about 90% of stage pairs have PCC below 0.1. However 9% of stage pairs have PCC above 0.5. This is perhaps because some of the stages run back-to-back on the same server when processing a request; if the server is slow for some reason, all the stages will be slow. However, in such cases, the reissued request is very likely to be sent to a different server. Hence, in spite of this mild correlation we treat the inherent processing latency across stages to be independent.

2.2.2 *Properties of execution DAGs*

As the “all stages” line in Fig. 6 indicates, most workflows have a lot of stages, with a median value of 14 and 90$^{\text{th}}$ percentile of 81. About 20% of the workflows have stage sequences of length 10 or more (not shown). However, the

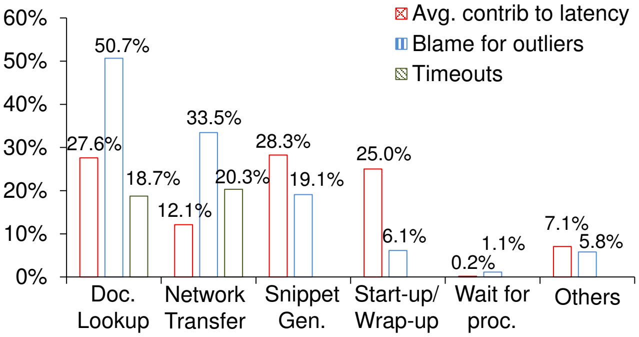


Figure 7: Stages that receive blame for the slowest 5% queries in the web-search workflow.

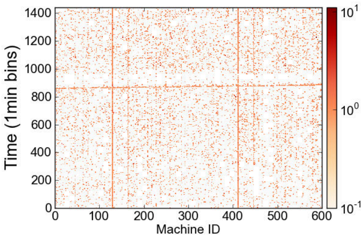


Figure 8: Heatmap showing how the latency varies across machines and time (for queries to the workflow in Fig 1).

max in-degree across stages is proportional to the number of stages in the workflow (not shown). That is, most workflows are parallel.

Stages that have a very small latency and rarely occur on the critical path for the query can be set aside. We say that *critical path* for a query is the sequence of dependent stages in the workflow that finished the last for that query. Since queries can have different critical paths, we consider the most frequently occurring critical paths that account for 90% of the queries. Along each critical path, we consider the smallest subset of stages that together account for over 90% of the query's latency and call these the *effective* stages. Fig. 6 plots the number of effective stages across workflows. We see that the number of effective stages in a workflow is sizable but much smaller than the number of all stages; median is 4 and 90^{th} percentile is 18. The figure also plots the average number of effective stages on these critical paths of workflows; median is 2.2. Finally, we plot a distribution of the in-degree of these effective stages on the critical paths; median is 2 and 90^{th} is 9. Hence, we see that production workflows even when counting only the effective stages are long and many way parallel.

We point out that stages with high in-degree, that aggregate responses from many other stages, are a significant source of variability. Whenever one of the input stages is slow, the output would be slow. In fact $P(max_n X_i > s) \sim nP(X_i > s)$ when s is in the tail. We see two types of such fan-in, only one of which (at the stage level) has been accounted for above. Several stages internally aggregate the responses from many servers, for example, the web-search stage above aggregates responses from 100s-1000s of servers that each retrieve documents matching the query from their shard of the index.

2.3 Causes for latencies on the tail

When responses take longer than typical, we want to understand the causes for the high latency. Here, we focus on the web-search workflow (see Fig. 1). For each of the 5% slowest queries, we assign blame to a stage when its contribution to that query's latency is more than $\mu + 2\sigma$, where μ, σ are the mean and stdev of its contribution over all queries. If a stage takes too long for a query, it is timed-out. In such cases, the blame is still assigned to the stage, citing timeout as the reason. Fig. 7 depicts, for each stage of the workflow, its average contribution to latency along with the fraction of delayed responses for which it has timed-out or is blamed (includes timeouts). Since more than one stage can be blamed for a delayed response, the blame fractions add up to more than one.

Parameter	Value Percentiles 50^{th}	90^{th}	99^{th}
at server, network load due to request-response traffic	895pps, .62Mbps	2242pps, 1.84Mbps	2730pps, 2.3Mbps
Lag to retransmit	67.2ms	113.3ms	168.7ms
Packet loss prob. of request-response traffic	.00443		
To compare: packet loss prob. of map-reduce	.0004336		
Fraction of losses recovered by RTO	.987		

Table 1: Network Characteristics

We see that the document lookup and the network transfer stages receive the most blame (50.7% and 33.5% each). In particular, these stages take so long for some queries that the scheduler times them out in 18.7% and 20.3% of cases respectively. Network transfer receives blame for many more outliers than would be expected given its typical contribution to latency (just 12.1%). We also see that though the start-up/ wrap-up stage contributes sizable average latency, it is highly predictable and rarely leads to outliers. Further, the servers are provisioned such that the time spent waiting in queues for processing at both the doc lookup and the snippet generation stages is quite small.

Why would stages take longer than typical? To examine the doc lookup stage further, we correlate the query latency with wall-clock time and the identity of the machine in the doc lookup tier that was the last to respond. Fig. 8 plots the average query latency per machine per second of wall time. The darkness of a point reflects the average latency on log scale. We see evidence of flaky machines in the doc lookup tier (dark vertical lines); queries correlated with these machines consistently result in higher latencies. We conjecture that this is due to hardware trouble at the server. We also see evidence for time-dependent events, i.e., periods when groups of machines slow down. Some are rolling upgrades through the cluster (horizontal sloping dark line), others (not shown) are congestion epochs at shared components such as switches. We also found cases when only machines containing a specific part of the index slowed down, likely due to trouble in parsing some documents in that part of the index.

To examine the network transfer stage further, we correlate the latency of the network transfer stage with packet-level events and the lag introduced in the network stack at either end. We collected several hours of packet traces in production beds for the network transfer stage in the web-search workflow (Fig. 1). To compare, we also collect packet traces from production map-reduce clusters that use the same server and switch hardware but carry traffic that is dominated by large flows. The results of this analysis is shown in Table 1.

We see that the request-response traffic has 10X higher loss rate than in the map-reduce cluster. Further the losses are bursty, coefficient of variation $\frac{\sigma}{\mu}$ is 2.4536. The increased loss rate is likely due to the scatter-gather pattern, i.e., responses collide on the link from switch to aggregator. Most of the losses are recovered only by a retransmission timeout (over 98%) because there are not enough acks for TCP's fast retransmission due to the small size of the responses. Surprisingly, the RTO for these TCP connections was quite large, in spite of RTO_min being set to 20ms; we are still investigating the cause. We conclude that TCP's inability to recover from burst losses for small messages is the reason behind the network contributing so many outliers.

2.4 Takeaways

Our analysis shows the following:

- Workflow DAGs are large and very complex, with significant sequences of stages and high degree of parallelism, which increases latency variance.
- Different stages have widely different properties of mean, variance, and variance as a function of the amount of requests reissued at that stage.
- Latencies of different stages are uncorrelated, except when running on the same machine; latency of reissues is uncorrelated with latency of the first request.

The first two observations demonstrate the complexity of the problem; heuristics that do not consider properties of the latency distributions and of the DAG, cannot perform very well. The third observation points in the direction of our solution; it allows us to decompose the optimization problem on a complex workflow to a problem over individual stages.

3. KEY IDEAS IN Kwiken

The goal of Kwiken is to improve the latency of request-response workflows, especially on the higher percentiles. We pick the variance of latency as the metric to minimize because doing so will speed-up all of the tail requests; in that sense, it is more robust than minimizing a particular quantile[1].

Our framework optimizes the workflow latency at both the stage and workflow levels. At the stage/local level, it selects a policy that minimizes the variance of the stage latency. At the workflow/global level, it combines these local policies to minimize the end-to-end latency. We employ three core per-stage techniques for latency reduction – reissue laggards at replicas, skip laggards to return timely albeit incomplete answers and *catch-up*, which involves speeding up requests based on their overall progress in the workflow.

Using latency reduction techniques incurs cost – such as using more resources to serve reissued requests – so we have to reason about apportioning a shared global cost budget across stages to minimize the end-to-end latency. For example, reissues have higher impact in stages with high variance. Similarly, speeding up stages that lie on the critical path of the workflow is more helpful than those that lie off the critical path. Also, as shown in Fig. 5, variance of some stages reduces quickly even with a few reissues, while other stages require more reissues to achieve the same benefits. Finally, the cost of reissuing the same amount of requests could be orders of magnitude higher in stages that are many-way parallel, a factor that has to be incorporated into the overall optimization.

To reason about how local changes impact overall latency, our basic idea is to decompose the variance of the workflow's latency into the variance of individual stages' latency. If the random variable L_s denotes the latency at stage s, then the latency of workflow w is given by

$$\mathcal{L}_w(L_1, \ldots, L_N) = \max_p \sum_{s \in p} L_s, \tag{1}$$

where p stands for a *path*, namely an acyclic sequence of stages through the workflow (from input to output). Ideally, we would use the variance of $\mathcal{L}_w$ as our objective function, and minimize it through allocating budget across stages. Unfortunately, however, the variance of $\mathcal{L}_w$ does not have a closed form as a function of the individual stages' statistics (e.g., their first or second moments). Instead, we resort to minimizing an upper bound of that variance. Recall from §2.2 that the different L_s can be roughly treated as independent random variables. Using this approximation together with (1) leads to the following decomposition:

$$\mathrm{Var}(\mathcal{L}_w) \leq \sum_{s \in w} \mathrm{Var}(L_s), \tag{2}$$

where $\mathrm{Var}(\cdot)$ denotes the variance of a random variable; see appendix for proof. The bound is always tight for sequential workflows, as stage variances add up. It can also be shown that (2) is the best general bound for parallel workflows; details omitted here.

Using Chebyshev's inequality, (2) immediately implies that $Pr(|\mathcal{L}_w - E\mathcal{L}_w| > \delta) \leq \frac{(\sum_s \mathrm{Var}(L_s))^2}{\delta^2}$. The bound indicates that minimizing the sum of variances is closely related to minimizing the probability of large latency deviations, or latency percentiles. Better concentration bounds (e.g., Bernstein [7]) can also be obtained; we omit details for brevity. We emphasize that we do not claim tightness of the bounds, but rather use them as a theoretical insight for motivating sum of variances minimization. As we elaborate below, the decomposition to sum of variances leads to a tractable optimization problem, unlike other approaches for solving it.

Alongside the latency goal, we need to take into account the overall cost from applying local changes. Here, we describe the framework using reissues. Formally, let r_s be the fraction of requests that are reissued at stage s and let c_s be the (average) normalized resource cost per request at stage s, i.e., $\sum_s c_s = 1$. Then the overall cost from reissues is $C(\mathbf{r}) = \sum_s c_s r_s$, and the problem of apportioning resources becomes:

$$\begin{aligned} \text{minimize} \quad & \sum_s \mathrm{Var}\big(L_s(r_s)\big) \\ \text{subject to} \quad & \sum_s c_s r_s \leq B, \end{aligned} \tag{3}$$

where B represents the overall budget constraint for the workflow[2] and $L_s(r_s)$ is the latency of stage s under a policy that reissues all laggards after a timeout which is chosen such that only an r_s fraction of requests are reissued. Since $\{c_s\}$ are normalized, B can be viewed as the fraction of additional

[1] Delaying responses such that all queries finish with the slowest has a variance of 0, but is not useful. An implicit requirement in addition to minimizing variance, which Kwiken satisfies, is for the mean to not increase.

[2] Considerations on how to choose the budget for each workflow, which we assume as given by an exogenous policy, are outside the scope of this paper.

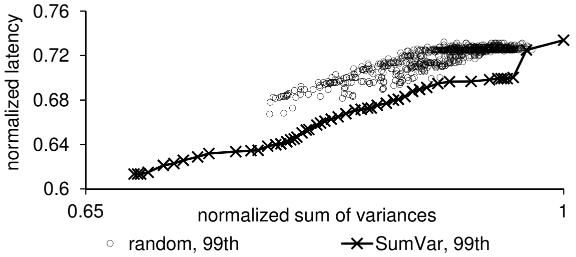


Figure 9: This plot shows the results of a random search in the reissue budget space (circles) and using SumVar on the same workflow (line with crosses). It illustrates that as sum of variances (x-axis, normalized) decreases, so does the 99th percentile of latency (y-axis, normalized).

resources used for latency reduction. The formulation in (3) can be generalized to accommodate multiple speedup techniques (see §4.4). This optimization problem is the basis for our algorithm.

We alternatively used a weighted version of the cost function, $\sum_s w_s \text{Var}\big(L_s(r_s)\big)$, where the weights w_s can be chosen to incorporate *global* considerations. For example, we can set w_s based on (i) total number of paths crossing stage s, or (ii) mean latency of s. However, our evaluation showed no significant advantages compared to the unweighted version.

We solve (3) by first constructing per-stage *models* (or *variance-response curves*), denoted $V_s(r_s)$ that estimate $\text{Var}(L_s(r_s))$ for each stage s. Due to the complexity of these curves (see Fig. 5), we represent them as empirical functions, and optimize (3) using an iterative approach based on gradient descent; see details in §4. Due to the non-convexity of (3), our algorithm has no performance guarantees. However, in practice, our algorithm already achieves adequate precision when the number of iterations is $O(N)$, where N is the number of stages in the workflow.

So far, we have considered "local" improvements, where latency reduction policy inside each stage is independent of the rest of the workflow. Our *catch-up* policies use the execution state of the entire request to make speed-up decisions. These are described in more detail in §4.3.

Finally, as described earlier, burst losses in the network are responsible for a significant fraction of high latencies. We recommend lowering RTO_min to 10ms and using a burst-avoidance technique such as ICTCP [26] at the application level. While not a perfect solution, it addresses the current problem and is applicable today.

4. DESIGN OF Kwiken

Here we provide details on applying our framework to the different techniques – reissues, incompleteness, and catch-up.

4.1 Adaptive Reissues

4.1.1 Per-stage reissue policies

Request reissue is a standard technique to reduce the latency tail in distributed systems at the cost of increasing resource utilization [10]. Reissuing a part of the workflow (i.e., reissuing the work at one or more servers corresponding to a stage) elsewhere in the cluster is feasible since services are often replicated and can improve latency by using the response that finishes first.

A typical use of reissues is to start a second copy of the request at time T_s, if there is no response before T_s, and use the first response that returns [10]. Given f_s, the latency distribution of stage s, and r_s, the target fraction of requests to reissue, the corresponding timeout T_s is equivalent to the $(1 - r_s)$ quantile of f_s. E.g., to reissue 5% of requests, we set T_s to the 95^{th} percentile of f_s. We can thus obtain the variance-response function $V_s(r_s)$ for different values of r_s, by computing the corresponding T_s, and then performing an offline simulation using the latencies from past queries at this stage. We use standard interpolation techniques to compute the convex-hull, $\bar{V}_s(r_s)$, to preclude discretization effects. Note that we can compute the variance-response function $V_s(r_s)$ for different reissue policies and pick the best one for each r_s (e.g., launching two reissues instead of just one after a timeout or reissuing certain fraction of requests right away, i.e., timeout of zero). With $\bar{V}_s(r_s)$, we note that our framework abstracts away the specifics of the per-stage latency improvements from the the end-to-end optimization. Further, $\bar{V}_s(r_s)$ needs to be computed only once per stage.

4.1.2 Apportioning budget across stages

Equipped with per-stage reissue policies captured in $\bar{V}_s(r_s)$, we apportion budget across stages by solving (3) with $\bar{V}_s(r_s)$ replacing $\text{Var}\big(L_s(r_s)\big)$ for every s.

Kwiken uses a greedy algorithm, SumVar, to solve (3) which is inspired by gradient descent. SumVar starts from an empty allocation ($r_s = 0$ for every stage)[3]. In each iteration, SumVar increases $r_{s'}$ of one stage s' by a small amount wherein the stage s' is chosen so that the decrease in (3) is maximal. More formally, SumVar assigns resources of cost $\delta > 0$ per iteration (δ can be viewed as the step-size of the algorithm). For each stage s, δ additional resources implies an increase in r_s by δ/c_s (since $c_s r_s$ = resource cost) which reduces variance by the amount $(\bar{V}_s(r_s) - \bar{V}_s(r_s + \delta/c_s))$. Hence, SumVar assigns δ resources to stage $s' \in \text{argmax}_s(\bar{V}_s(r_s) - \bar{V}_s(r_s + \delta/c_s))$; ties are broken arbitrarily. Returning to the example in Fig. 2, SumVar would allocate budget to Stage 1 in the early steps, since its variance decreases rapidly even with a small budget. As the variance of Stage 1 flattens out, subsequent steps would allocate budget to Stage 2. The algorithm converges to an allocation of 1:3 when all budget has been allocated thereby minimizing the sum of the variances of the stages, and in turn the 99^{th} latency percentile.

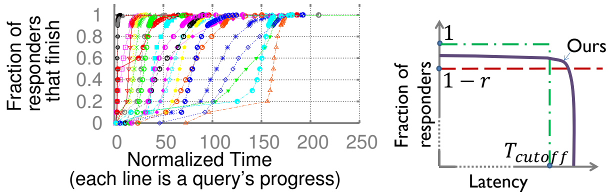
We demonstrate our algorithm on a production workflow with 28 stages and significant sequential and parallel components. First, we generate 1000 random reissue allocation and for each, plot the achieved sum of variances and 99^{th} percentile of end-to-end latency (circles in Fig. 9). Notice that as sum of variances decreases, so does the latency percentile. This illustrates that the intuition behind our approach is correct; even in complex workflows, sum of variances is a good metric to minimize in order to improve tail latencies. Second, we plot the progress of our algorithm on the same workflow (line with crosses in Fig. 9, from right to left). It shows that the gradient descent approach can achieve lower latency than a random search.

Our experiments show that it suffices to divide the budget into γN chunks of equal size, where $\gamma \in [2, 4]$ and N is the number of stages. Consequently, the number of iterations is linear in the number of stages. Each iteration requires

[3] In our experiments, we tried other initial allocations, but they did not improve performance.

latency distribution	//'ism (n)	utility loss (r)	Latency Reduction (% over baseline) 95th	99th
Normal mean=1, sd=10	10000	.001	25.33%	29.22%
		.01	44.29%	47.67%
LogNormal meanlog=1, sdlog=2	10000	.001	90.34%	93.83%
		.01	98.22%	98.96%
LogNormal meanlog=1, sdlog=2	1000	.001	59.93%	64.71%
		.01	93.30%	96.12%
Web	1000s	.001	4.1%	4.0%
		.01	43.1%	77.7%
Image	100s	.001	0%	0%
		.01	42.6%	81.2%
Video	100s	.001	0%	0%
		.01	31.2%	51.3%

Table 2: Given utility loss rate (r), the improvement in latency from stopping when the first $\lceil n(1-r)\rceil$ responders finish.



(a) Progress of a few sample queries for the web workflow **(b) Comparing strategies**

Figure 10: Trading off Incompleteness for Latency

$O(\log N)$ for recomputing a single gradient component and inserting it into a heap structure. Consequently, the overall complexity of our algorithm is $O(N \log N)$. Importantly, it is independent of the topology of the DAG.

4.2 Trading off Completeness for Latency

In many situations, partial answers are useful both to the user and to subsequent stages in the workflow. An example is a workflow which picks the best results from many responders. Similar to the web-search stage shown in Fig. 1, the image-, video- and ad- search stages at Bing consist of many servers that in parallel compute the best matching images, videos, and ads for a search phrase. In each, an aggregator picks the best few responses overall. Not waiting until the last server responds will speed up the workflow while returning an incomplete response.

How to measure the usefulness of an incomplete answer? Some stages have explicit indicators; e.g., each returned document in the web-search workflow has a relevance score. For such stages, we say that the answer has utility 1 if it contains the highest ranked document and 0 otherwise (alternatively, we can give weights to each of the top ten results). For other stages, we let utility be the fraction of components that have responded[4]. We define the utility loss of a set of queries as the average loss of utility across all the queries. So, if 1 out of 1000 queries loses the top ranked document, the utility loss incurred is 0.1%.

Discussions with practitioners at Bing reveal that enormous developer time goes into increasing answer relevance by a few percentage points. While imprecise answers are acceptable elsewhere (e.g., in approximate data analytics), we conclude that request-response workflows can only afford a small amount of utility loss to improve latency. Hence Kwiken works within a strict utility loss rate of .001 (or 0.1%).

Our goal is to minimize response latency given a constraint on utility loss. To be able to use (in)completeness as a tool in our optimization framework, we treat utility loss as a "resource" with budget constraint (average quality loss) and decide how to apportion that resource across stages and across queries within a stage so as to minimize overall latency variance. We emphasize that this formulation is consistent with Bing; both reduction in latency and higher quality answers improve user satisfaction and can be used interchangeably in optimization.

4.2.1 *Using incompleteness within a stage*

The basic setup here is of a single stage, with a constraint on the maximal (expected) quality loss rate, denoted r. Consider a simple strategy: Let each query run until $\lceil n(1-r)\rceil$ of its n responders return their answer. Then if the best document's location is uniform across responders, the expected utility loss is r. To appreciate why it reduces latency, consider a stage with n responders whose latencies are $X_1 \dots X_n$, this strategy lowers query latency to the $\lceil n(1-r)\rceil$'th largest value in $X_1, \dots X_n$ as opposed to the maximum X_i.

Table 2 shows the latency reductions for a few synthetic distributions and for the web, image, and video search stages where we replay the execution traces from tens of thousands of production queries at Bing. First, we note that even small amounts of incompleteness yield disproportionately large benefits. For a normal distribution with mean 1 and stdev 10, we see that the 95$^{\text{th}}$ percentile latency reduces by about 25% if only 0.1% of the responses are allowed to be incomplete. This is because the slowest responder(s) can be much slower than the others. Second, all other things equal, latency gains are higher when the underlying latency distribution has more variance, or the degree of parallelism within the stage is large. LogNormal, a particularly skewed distribution, has 3.5X larger gains than Normal but only 2.2X larger when the number of parallel responders drops to 10^3. However, we find that the gains are considerably smaller for our empirical distributions. Partly, this is because these distributions have bounded tails, since the user or the system times-out queries after some time.

To understand why the simple strategy above does not help in practice, Fig. 10 (a) plots the progress of example queries from the web-search stage. Each line corresponds to a query and shows the fraction of responders vs. elapsed time since the query began. We see significant variety in progress– some queries have consistently quick or slow responders (vertical lines on the left and right), others have a long tail (slopy top, some unfinished at the right edge of graph) and still others have a few quick responders but many slow ones (steps). To decide which queries to terminate early (subject to overall quality constraint), one has to therefore take into account *both* the progress (in terms of responders that finished) and the elapsed time of the individual query. For example, there are no substantial latency gains from early termination of queries with consistently quick responders, as even waiting for the last responder may not impact tail latency. On the other hand, a slow query may be worth terminating even before the bulk of responders complete.

Building up on the above intuition, Kwiken employs dynamic control based on the progress of the query. Specifically, Kwiken terminates a query when either of these two condi-

[4] If responders are equally likely to yield the most relevant result, these measures yield the same expected loss.

tions hold: i) the query has been running for T_d time after p fraction of its components have responded, ii) the query runs for longer than some cut-off time T_c. The former check allows Kwiken to terminate a query based on its progress, but not terminate too early. The latter check ensures that the slowest queries will terminate at a fixed time regardless of however many responders are pending at that time. Fig. 10 (b) visually depicts when queries will terminate for the various strategies.

Kwiken chooses these three parameters empirically based on earlier execution traces. For a given r, Kwiken parameter sweeps across the (T_c, T_d, p) vectors that meet the quality constraint with equality, and computes the variance of stage latency. Then, Kwiken picks the triplet with the smallest variance. Repeating this procedure for different values of r yields the variance-response curve $V(r)$ (cf. §3). Note that the approach for obtaining $V(r)$ is data driven. In particular, the choice of parameters will vary if the service software is rewritten or the cluster hardware changes. From analyzing data over an extended period of time, we see that parameter choices are stable over periods of hours to days (we show results with disjoint test and training sets in §5.3).

4.2.2 *Composing incompleteness across stages*

Any stage that aggregates responses from parallel components benefits from trading off completeness for latency. When a workflow has multiple such stages, we want to apportion utility loss budget across them so as to minimize the end-to-end latency. The approach is similar in large part to the case of reissues – the variance-response curves computed for each stage help split the overall optimization to the stage-level.

Unlike reissue cost, which adds up in terms of compute and other resources, utility loss is harder to compose, especially in general DAGs where partial results are still useful. Modeling such scenarios fully is beyond the scope of this paper. Nevertheless, we show two common scenarios below, where the budget constraint can be written as a weighted sum over the loss rates at individual stages. First, consider a sequential workflow with N stages where an answer is useful only if every stage executes completely. If r_i is the loss budget of stage i, the overall utility loss is given by $r_1 + (1-r_1)r_2 + \cdots + \big(\prod_{s=1}^{N-1}(1-r_s)\big)r_N$. which is upper bounded by $\sum_i r_i$. Hence, the budget constraint is $\sum_i r_i \le B$, i.e., the "cost" of loss c_s is one at all stages. Second, consider stages that are independent in the sense that the usefulness of a stage's answer does not depend on any other stage (e.g., images and videos returned for a query). Here, the overall utility loss can be written as $\frac{\sum r_s u_s}{\sum u_s}$ where u_s is the relative contribution from each stage's answer. Then, the budget constraint is given by $\sum_s c_s r_s \le B$, where $c_s = \frac{u_s}{\sum_{s'} u_{s'}}$.

4.3 Catch-up

The framework described so far reduces workflow latency by making local decisions in each stage. Instead, the main idea behind catch-up is to speed-up a request based on its progress in the workflow as a whole. For example, when some of the initial stages are slow, we can reissue a request more aggressively in the subsequent stages. In this paper, we consider the following techniques for catch-up: (1) allocate more threads to process the request; given multi-threaded implementation of many stages at Bing, we find that allocating more threads to a request reduces its latency. (2) Use high-priority network packets on network switches for lagging requests to protect them from burst losses. And (3), reissue requests more aggressively based on the total time spent in the workflow – we call this *global reissues* to distinguish it from local reissues (discussed in §4.1), where the reissue is based on time spent within a stage.

Each of these techniques uses extra resources and could affect other requests if not constrained. To ensure catch-up does not overload the system, Kwiken works within a catch-up budget per workflow. Given per-stage budget, Kwiken estimates a threshold execution latency, T_z, and speeds the parts of a query that remain after T_z using the techniques above. Since the decisions of a catch-up policy depend on request execution in previous stages, allocating catchup budget across stages cannot be formulated as a separable optimization problem, unlike the case of local techniques (§4.1.2). We therefore use simple rules of thumb. For example, for global reissues, we allocate catch-up budget proportionally to the budget allocation for local reissues. Intuitively, a stage that benefits from local reissues, can also speed-up lagging requests. We evaluate the catch-up policies in §5.4.

4.4 Putting it all together

To conclude, we briefly highlight how to combine different techniques into a unified optimization framework. Using superscripts for the technique *type*, let $k = 1, \ldots, K$ be the collection of techniques, then our optimization framework (3) extends as follows to multiple dimensions:

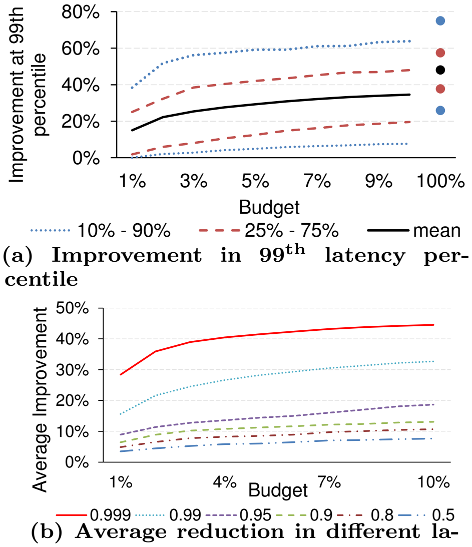
$$\begin{aligned} \text{minimize} \quad & \sum_s \mathrm{Var}\big(L_s(r_s^1, \ldots, r_s^K)\big) \\ \text{subject to} \quad & \sum_s c_s^k r_s^k \le B^k, \quad k = 1, \ldots, K. \end{aligned} \tag{4}$$

As before, $\mathrm{Var}\big(L_s(r_s^1, \ldots, r_s^K)\big)$ are the per-stage models, i.e., variance-response curves. These models abstract away the internal optimization given $(r_s^1, \ldots, r_s^K)$. Greedy gradient-like algorithms (such as SumVar) can be extended to solve (4), however, the extension is not straightforward. The main complexity in (4) is hidden in the computation of the variance-response curves – as opposed to a scan over one dimension in (3), variance-response curves in (4) requires a scan over the k-dimensional space, $(r_s^1, \ldots, r_s^K)$. In practice, we note that the optimization often decouples into simpler problems. For example, assume $K = 2$ with reissues and partial responses as the two techniques for reducing latency. Partial responses are mostly useful in many-way parallel services which have a high cost for reissues. Hence, we can use the utility loss budget only for the services with high fan-out and the reissue budget for the rest of the services. Finding a general low complexity algorithm to solve (4) is left to future work.

5. EVALUATION

In this section, we evaluate the individual techniques in Kwiken by comparing them to other benchmarks (§5.2 - §5.4), followed by using all Kwiken techniques together (§5.5). Using execution traces and workflows from Bing, we show that:

- With a reissue budget of just 5%, Kwiken reduces the 99th percentile of latency by an average of 29% across workflows. This is over half the gains possible from reissuing every request (i.e., budget=100%). Kwiken's



(a) Improvement in 99th latency percentile

(b) Average reduction in different latency percentiles

Figure 11: Reduction in latency achieved by using per-stage reissues in Kwiken

apportioning of budget across stages is key to achieving these gains.

- In stages that aggregate responses from many responders, Kwiken improves the 99th percentile of latency by more than 50% with a utility loss of at most 0.1%.
- Using simple catch-up techniques, Kwiken improves the 99th percentile latency by up to 44% by using just 6.3% more threads and prioritizing 1.5% network traffic.
- By combining reissues with utility trade-off we see that Kwiken can do much better than when using either technique by itself; for example, a reissue budget of 1% combined with a utility loss of 0.1% achieves lower latency than just using reissues of up to 10% and just trading off utility loss of up to 1%.
- Kwiken's data-driven parameter choices are stable.

5.1 Methodology

Traces from production: To evaluate Kwiken, we extract the following data from production traces for the 45 most frequently accessed workflows at Bing: workflow DAGs, request latencies at each stage as well as the end-to-end latency, the cost of reissues at each stage and the usefulness of responses (e.g., ranks of documents) when available. To properly measure latencies on the tail, we collected data for at least 10,000 requests for each workflow and stage. The datasets span several days during Oct-Dec 2012. We ignore requests served from cache at any stage in their workflow; such requests account for a sizable fraction of all requests but do not represent the tail.

We conducted operator interviews to estimate the cost of reissue at each stage. Our estimates are based on the resources expended per request. For stages that process the request in a single thread, we use the mean latency in that stage as an approximation to the amount of computation and other resources used by the request. For more complex stages, we use the sum of all the time spent across parallel servers to execute this stage. Kwiken only relies on the relative costs across stages when apportioning budget.

Simulator: We built a trace-driven simulator, that mimics the workflow controller used in production at Bing, to test the various techniques in Kwiken. The latency of a request at each stage and that of its reissue (when needed) are sampled independently from the distribution of all request latencies at that stage. We verified that this is reasonable: controlled experiments on a subset of workflows where we issued the same request twice back-to-back showed very small correlation; also, the time spent by a request in different stages in its workflow had small correlation (see §2.2).

Estimating Policy Parameters: The parameters of the Kwiken policies (such as per-stage reissue timeouts) are trained based on traces from prior executions. While we estimate the parameters on a *training data set*, we report performance of all polices on a *test data set* collected at a different period of time. In all cases, both training and test data sets contain traces from several thousands to tens of thousands of queries.

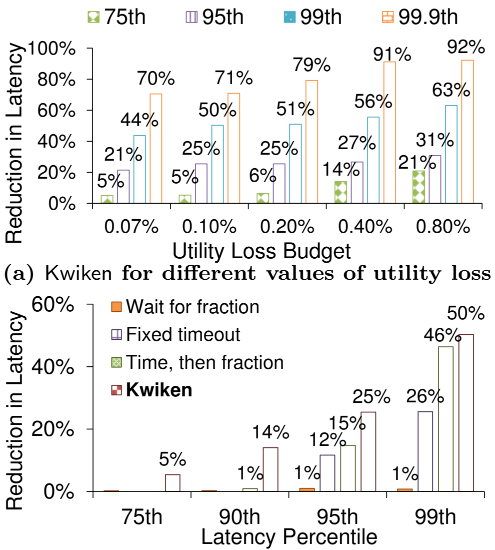
5.2 Reissues

We first evaluate the effect of using per-stage reissues within Kwiken's framework. Fig. 11a plots Kwiken's improvements on the end-to-end latency due to reissues, using the SumVar algorithm described in §4.1. The x-axis depicts the fraction of additional resources provided to reissues and the y-axis shows the fractional reduction at the 99th percentile. The solid line shows the mean improvement over the 45 most frequent workflows at Bing; the dashed lines represent the spread containing the top 25% and bottom 75% of workflows and the dotted lines show the improvements for the top 10% and bottom 90% of workflows (sorted with respect to percentage improvement). The circles on the right edge depict latency improvements with a budget of 100%.

We see that Kwiken improves 99th percentile of latency by about 29%, on average, given a reissue budget of 5%. This is almost half the gain that would be achieved if all requests at all stages were reissued, i.e., a budget of 100%. This indicates that Kwiken ekes out most of the possible gains, i.e., identifies laggards and tries to replace them with faster reissues, with just a small amount of budget. Most workflows see gains but some see a lot more than the others; the top 10% of workflows improve by 55% while the top 75% of workflows see at least 10% improvement each. The variation is because different workflows have different amounts of inherent variance.

Fig. 11b plots the average gains at several other latency percentiles. As before, we see that small budgets lead to sizable gains and the marginal increase from additional budget is small. This is because some stages with high variance and low cost can be reissued at substantial fraction (e.g., 50%), yet consume only a fraction of total budget. It is interesting though that just a small amount of budget (say 3%) leads to some gains at the median. Observe that higher percentiles exhibit larger improvements, which is consistent with theory (cf. §3). We note that Kwiken scales efficiently to large workflows. Computing per-stage models takes about 2 seconds per stage and is parallelizable. Running SumVar takes less than one second for most of the workflows.

Comparing SumVar to other benchmarks: First, we compare against the current reissue strategy used in Bing. The actual reissue timeout values used in Bing are very conservative – additional cost is only 0.2% – and reduce 99th percentile of latency only by 3%. The timeouts are so conservative because without an end-to-end framework such as



(a) Kwiken **for different values of utility loss**

(b) Kwiken **vs. other variants for utility loss** $= 0.1\%$

Figure 12: Latency reductions achieved by Kwiken **and variants when trading off completeness for latency.**

Kwiken, it is hard to reason about how much of the overall capacity should be used for reissues at each stage.

Second, we compared with several straw man policies. One policy would assign each stage the same reissue fraction $r_i = r$. However such policy has clear shortcomings; for example, if a single stage has high cost c_i it will absorb most of the budget. If that stage has low variance, then the resulting end to end improvement will be negligible. Other policies like allocating equal budget to each stage exhibit similar drawbacks.

Finally, lacking an optimal algorithm (recall that even (3) has a non-convex objective), we compare with two brute-force approaches. For a subset of nine smaller workflows and budgets from 1% to 10%, we pick the best timeouts out of $10,000$ random budget allocations. Compared to training Kwiken on the same data, this algorithm was about 4 orders of magnitude slower. Hence, we did not attempt it on larger workflows. Here, Kwiken's results were better on average by 2%; in 95% of the cases (i.e., $\{$workflow, budget$\}$ pairs), Kwiken's latency reduction was at least 94% of that achieved by this method. The second approach uses gradient-descent to directly minimize the 99^{th} percentile of the end-to-end latency using a simulator (i. e., avoiding the sum of variances approximation). This method was equally slow and performed no better. Hence, we conclude that Kwiken's method to apportion budget across stages is not only useful but also perhaps nearly as effective as an ideal (impractical) method.

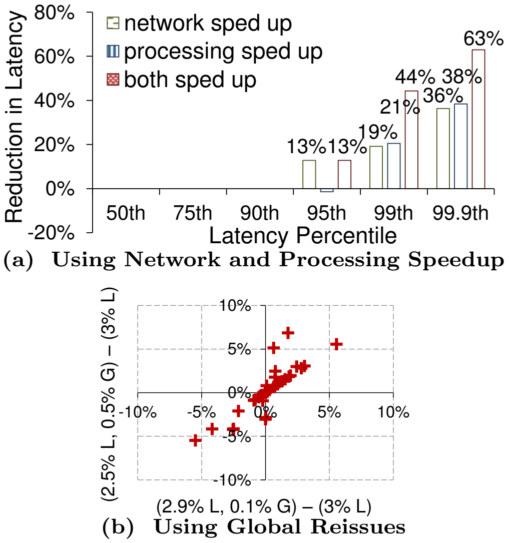
We also evaluated two weighted forms of Kwiken that more directly consider the structure of the workflow (§3): weighting each stage by its average latency and by its likelihood to occur on a critical path. While both performed well, they were not much better than the unweighted form for the examined workflows.

5.3 Trading off Completeness for Latency

Next, we evaluate the improvements in latency when using Kwiken to return partial answers. Fig. 12a plots the improvement due to trading off completeness for latency for different values of utility loss. Recall that our target is to be complete enough that the best result is returned for over 99.9% of the queries, i.e., a utility loss of 0.1%. With that budget, we see that Kwiken improves the 99^{th} (95^{th}) percentile by around 50% (25%). The plotted values are averages over the web, image and video stages. Recall that these stages issue many requests in parallel and aggregate the responses (see Fig. 1).

Fig. 12b compares the performance of Kwiken with a few benchmarks for utility loss budget of 0.1%: *wait-for-fraction* terminates a query when b fraction of its responders return, *fixed-timeout* terminates queries at T_{cutoff}, and *time-then-fraction* terminates queries when both these conditions hold: a constant T' time has elapsed and at least α fraction of responders finish.

We see that Kwiken performs significantly better. *Wait-for-fraction* spends significant part of budget on queries which get the required fraction relatively fast, hence slower queries that lie on the tail do not improve enough. *Fixed-timeout* is better since it allows slower queries to terminate when many more of their responders are pending but it does not help at all with the quick queries– no change below the 90^{th} percentile. Even among the slower queries, it does not distinguish between queries that have many more pending responders and hence a larger probability of losing utility versus those that have only a few pending responders. *Time-then-fraction* is better for exactly this reason; it never terminates queries unless a minimal fraction of responders are done. However, Kwiken does even better; by waiting for extra time after a fraction of responders are done it provides gains for both the quicker queries and variable amounts of waiting for the slower queries. Also, it beats *time-then-fraction* on the slowest queries by stopping at a fixed time.



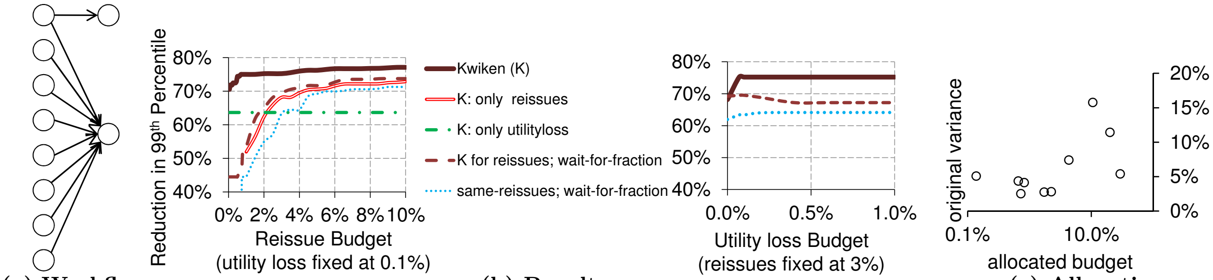
(a) **Using Network and Processing Speedup**

(b) **Using Global Reissues**

Figure 13: Latency improvements from using different catch-up techniques.

5.4 Catch-up

Here, we estimate the gains from the three types of catch-up mechanisms discussed earlier. Fig. 13a shows the gains of using multi-threaded execution and network prioritization on the web-search workflow (Fig. 1), relative to the baseline where no latency reduction techniques are used. We note that the speedup due to multi-threading is not linear with the number of threads due to synchronization costs and using 3 threads yields roughly a 2X speed up. We see that speeding



(a) Workflow (b) Results (c) Allocation

Figure 14: A detailed example to illustrate how Kwiken works

up both stages offers much more gains than speeding up just one of the stages; the 99th percentile latency improves by up to 44% with only small increases in additional load – about 6.3% more threads needed and about 1.5% of the network load moves into higher priority queues.

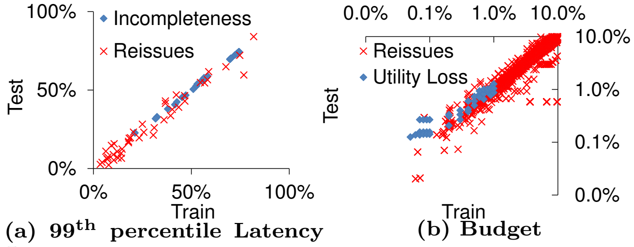
Next, we evaluate the usefulness of using global reissues on workflows. Using a total reissue budget of 3%, Fig. 13b plots the marginal improvements (relative to using the entire budget for local reissues) from assigning $\frac{1}{30}$th (x-axis) vs. assigning $\frac{1}{6}$th of the budget to global reissues (y-axis), for the 45 workflows we analyze. The average reduction in 99th percentile latency is about 3% in both cases, though, assigning $\frac{1}{6}$ of budget leads to higher improvements in some cases. Overall, 37 out of the 45 workflows see gains in latency. We end by noting that this experiment only shows one way to assign global reissues; better allocation techniques may yield larger gains.

5.5 Putting it all together

To illustrate the advantage of using multiple latency reduction techniques in the Kwiken framework together, we analyze in detail its application to a major workflow in Bing that has 150 stages. A simplified version of the workflow with only the ten highest-variance stages is shown in Fig. 14a. In three of the stages, we use utility loss to improve latency and use reissues on all stages.

We compare Kwiken with other alternatives in Fig. 14b; on left, we fix utility loss budget at 0.1% and vary reissues, on right, we vary utility loss and fix reissues at 3%. We see complementary advantage from using reissues and utility loss together. In the left graph, using Kwiken with reissues only at 10% performs worse than using both reissues at 1% and 0.1% utility loss. Also, using both together is about 20% better than using just utility loss. Graph on the right shows that, with reissue budget at 3%, increasing utility loss has very little improvements beyond .1%. We observe that the larger the reissue budget, the larger the amount of utility loss that can be gainfully used (not shown). Further, how we use the budget also matters; consider *K for reissues; wait-for-fraction* on the left. For the same amount of utility loss, Kwiken achieves much greater latency reduction.

So what does Kwiken do to get these gains? Fig. 14c shows for each of the ten stages, the latency variance (as a fraction of all variance in the workflow) and the amount of allocated budget (in log scale). We see that the budget needs to be apportioned to many different stages and not simply based on their variance, but also based on the variance-response curves and the per-stage cost of request reissue. Without Kwiken, it would be hard to reach the correct assignment.



(a) 99th percentile Latency Improvement (b) Budget

Figure 15: For 45 workflows, this figure compares the 99th percentile latency improvement and budget values of the training and test datasets.

5.6 Robustness of parameter choices

Recall that Kwiken chooses its parameters based on traces from prior execution. A concern here is that due to temporal variations in our system, the chosen parameters might not yield the gains that are expected from our optimization or may violate resource budgets.

To understand the stability of the parameter choices over time, we compare the improvements for 99th percentile latency and the budget values obtained for the "training" dataset, to those obtained for the "test" datasets. The test datasets were collected from the same production cluster on three different days within the same week. Fig. 15 shows that the latency improvements on the test datasets are within a few percentage points off that on the training datasets. The utility loss on the test dataset is slightly larger but predictably so, which allows us to explicitly account for it by training with a tighter budget. In all, we conclude that Kwiken's parameter choices are stable. Also, reallocating budget is fast and can be done periodically whenever the parameters change.

6. RELATED WORK

Improving latency of datacenter networks has attracted much recent interest from both academia and industry. Most work in this area [2, 3, 17, 25, 29] focuses on developing transport protocols to ensure network flows meet specified deadlines. Approaches like Chronos [18] modify end-hosts to reduce operating system overheads. Kwiken is complementary to these mechanisms, which reduce the latency of individual stages, because it focuses on the end-to-end latency of distributed applications.

Some recent work [4, 13, 28] reduces the job latency in (MapReduce-like) batch processing frameworks [11] by adaptively reissuing tasks or changing resource allocations. Other prior work [19] explores how to (statically) schedule jobs, that are modeled as a DAG of tasks to minimize completion time. Neither of these apply directly to the context of Kwiken which targets large interactive workflows that finish within a few hundreds of milliseconds and may involve over thousands of servers. Static scheduling is relatively easy here and there is too little time to monitor detailed aspects at runtime (e.g., task progress) which was possible in the batch case.

Some recent work concludes that latency variability in cloud environments arises from contention with co-located services [1] and provides workload placement strategies to avoid interference [27].

Some of the techniques used by Kwiken have been explored earlier. Reissuing requests has been used in many distributed systems [10, 12] and networking [5, 14, 23] scenarios. Kwiken's contribution lies in strategically apportioning reissues across the stages of a workflow to reduce end-to-end latency whereas earlier approaches consider each stage independently. Partial execution has been used in AI [30] and programming languages [6, 16]. The proposed policies, however, do not translate to the distributed services domain. Closer to us is Zeta [15], which devises an application-specific scheduler that runs beside the query to estimate expected utility and to choose when to terminate. In contrast, Kwiken relies only on opaque indicators of utility and hence the timeout policies are more generally applicable.

7. CONCLUSION

In this paper, we propose and evaluate Kwiken, a framework for optimizing end-to-end latency in computational workflows. Kwiken takes a holistic approach by considering end-to-end costs and benefits of applying various latency reduction techniques and decomposes the complex optimization problem into a much simpler optimization over individual stages. We also propose novel policies that trade off utility loss and latency reduction. Overall, using detailed simulations based on traces from our production systems, we show that using Kwiken, the 99th percentile of latency improves by over 75% when just 0.1% of the responses are allowed to have partial results and 3% extra resources are used for reissues.

Acknowledgements: We thank Junhua Wang, Navin Joy, Eric Lo, and Fang Liu from Bing for their invaluable help. We thank Yuval Peres, our shepherd Georgios Smaragdakis and the SIGCOMM reviewers for feedback on earlier drafts of the paper.

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *Hotnets*, 2012.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao. Improving web availability for clients with MONET. In *NSDI*, 2005.

[6] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *PLDI*, 2010.

[7] S. Boucheron, G. Lugosi, and O. Bousquet. Concentration inequalities. *Advanced Lectures on Machine Learning*, 2004.

[8] J. Brutlag. Speed matters for Google web search. http://googleresearch.blogspot.com/2009/06/speed-matters.html, 2009.

[9] J. R. Dabrowski and E. V. Munson. Is 100 Milliseconds Too Fast? In *CHI*, 2001.

[10] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] G. Decandia et al. Dynamo : Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[13] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[14] D. Han, A. Anand, A. Akella, and S. Seshan. RPT: Re-architecting Loss Protection for Content-Aware Networks. In *NSDI*, 2012.

[15] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling Interactive Services with Partial Execution. In *SOCC*, 2012.

[16] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *ASPLOS*, 2011.

[17] C. Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

[18] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC*, 2012.

[19] Y. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys (CSUR)*, 1999.

[20] R. Nishtala et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[21] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *OSDI*, 2012.

[22] E. Schurman and J. Brutlag. The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search. http://velocityconf.com/velocity2009/public/schedule/detail/8523, 2009.

[23] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is Less: Reducing Latency via Redundancy. In *HotNets*, 2012.

[24] X. S. Wang et al. Demystifying Page Load Performance with WProf. In *NSDI*, 2013.

[25] C. Wilson et al. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[26] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CONEXT*, 2010.

[27] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[28] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[29] D. Zats et al. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.

[30] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.

APPENDIX

Proof of (2)**.** For each random variable L_s we introduce a new independent random variable L'_s which has the same distribution as L_s. Let $\mathbf{L} = (L_1, \ldots, L_N)$ and $\mathbf{L}^{(s)} = (L_1, \ldots, L_{s-1}, L'_s, L_{s+1}, \ldots, L_N)$. Then, using the Efron-Stein inequality [7], we have $\mathrm{Var}(\mathcal{L}_w(\mathbf{L})) \leq \frac{1}{2} \sum_s E\big[(\mathcal{L}_w(\mathbf{L}) - \mathcal{L}_w(\mathbf{L}^{(s)}))^2\big] \leq \frac{1}{2} \sum_s E\big[(L_s - L'_s)^2\big] = \sum_s \mathrm{Var}(L_s)$. □