Stout: An Adaptive Interface to Scalable Cloud Storage

John C. McCullough, John Dunagan*, Alec Wolman*, and Alex C. Snoeren

UC San Diego and *Microsoft Research, Redmond

ABSTRACT

Many of today's applications are delivered as scalable, multi-tier services deployed in large data centers. These services frequently leverage shared, scale-out, key-value storage layers that can deliver low latency under light workloads, but may exhibit significant queuing delay and even dropped requests under high load.

Stout is a system that helps these applications adapt to variation in storage-layer performance by treating scalable key-value storage as a shared resource requiring congestion control. Under light workloads, applications using Stout send requests to the store immediately, minimizing delay. Under heavy workloads, Stout automatically batches the application's requests together before sending them to the store, resulting in higher throughput and preventing queuing delay. We show experimentally that Stout's adaptation algorithm converges to an appropriate batch size for workloads that require the batch size to vary by over two orders of magnitude. Compared to a non-adaptive strategy optimized for throughput, Stout delivers over 34× lower latency under light workloads; compared to a non-adaptive strategy optimized for latency, Stout can scale to over 3× as many requests.

1. INTRODUCTION

Application developers are increasingly moving towards a software-as-a-service model, where applications are deployed in data centers and dynamically accessed by users through lightweight client interfaces, such as a Web browser. These "cloud-based" applications may run on hundreds or even thousands of servers to support hundreds of millions of users; the application servers in turn leverage high-performance scalable key-value storage systems, such as Google's BigTable [7] and Microsoft's Azure Storage [3], that allow them to gracefully handle variable client demand. Unfortunately, because these storage systems support many applications on a single shared infrastructure, they present application developers with a new source of variability: every application must now cope with a store that is being loaded by many applications' changing workloads.

Unlike variability in its own workload, which an application can easily monitor and often even predict, changes in the level of competition for shared storage resources are likely to be unexpected and outside the control of a particular application. Instead, each individual application must observe and react to changes in available storage-system throughput. Ideally, the collection of applications leveraging a particular scalable storage system would cooperate to achieve a mutually beneficial operating point that neither overloads the storage system nor starves any individual application.

Today, each application seeks to minimize its own perceived latency by sending each storage request immediately. Each storage request thus incurs overheads such as networking delay, protocol-processing, lock acquisitions, transaction log commits, and/or disk scheduling and seek time. However, when the store becomes heavily loaded, sending each request individually can lead to queuing at the store, and consequently high delay or even loss due to timeouts. In such heavily loaded situations, the throughput of the storage service can often be improved by batching multiple requests together, thereby reducing queuing delay and loss. Batching achieves this improvement by amortizing the previously mentioned overheads across larger requests, and prior work has documented that many stores provide higher throughput on larger requests [7, 16, 35].

Dynamically adjusting their degree of batching allows applications to achieve lower latency under light load and higher throughput under heavy load. Unfortunately, existing work applying control theory to computer systems offers no easily applicable solutions [18, 23]. For example, a common assumption in control theory is *modest actuation delay*: a reasonable and known fixed time between when an application changes its request rate and the store responds to this change. Scale-out key-value storage systems do not have such bounds, as an application can easily create a very deep pipeline of requests to the storage system. Other control theory techniques avoid this assumption, but bring other assumptions that are similarly unsatisfied by such storage systems. Instead, we observe that managing independent application demands in a scale-out key-value storage environment is quite similar to congestion control in a network: the challenge in both settings is determining an application's (sender's) "fair share." Moreover, the constraints of distributed congestion control—multiple, independent agents, unbounded actuation delay, and lack of a known bandwidth target—are quite similar to our own. Hence, we take inspiration from CTCP [37], a recently proposed delay-based TCP variant which updates send-rates based on deviation from the measured round-trip latency.

We propose an adaptive interface to cloud key-value storage layers, called Stout, that implements distributed congestion control for client requests. Stout works without any explicit information from the storage layer: its adaptation strategy is implemented solely at the application server (the storage client) and is based exclusively on the measured latency from unmodified scalable storage systems. This allows Stout to be more easily deployed, as individual cloud applications can adopt Stout without changing the shared storage infrastructure. Stout both adapts to sudden changes in application workload and converges to fairness among multiple, competing application servers employing Stout.

We show experimentally that Stout delivers good performance across a range of workloads requiring batching intervals to vary by over two orders of magnitude, and that Stout significantly outperforms any strategy using a fixed batching interval. Based on these results, Stout demonstrates that much of the benefit of adaptation can be had without needing to modify existing storage systems; to use a new store, Stout requires only internal re-calibration. By allowing cloud applications to sustain higher request rates under bursts, Stout can help reduce the expense of over-provisioning [8, 34]. Simultaneously, Stout provides good common case storage latency; this is critical to user-perceived latency because generating a user response often requires multiple interactions with the storage layer, thereby incurring this latency multiple times [11].

The primary novelty of Stout is its adaptive algorithm for dynamically adjusting the batching of storage requests. To better understand both the benefits and challenges in building an adaptive interface to shared cloud storage, we evaluate our adaptive control loop using a workload inspired by a real-world cloud service that is one component of Microsoft's Live Mesh cloud-based synchronization service [27]. In our performance evaluation, we demonstrate that: 1) Stout successfully adapts to a wide range of offered loads, providing under light workloads over 34× lower latency than a long fixed batching interval optimized for throughput, and under heavy workloads over 3× the throughput of a short fixed batching interval optimized for latency; 2) Stout provides

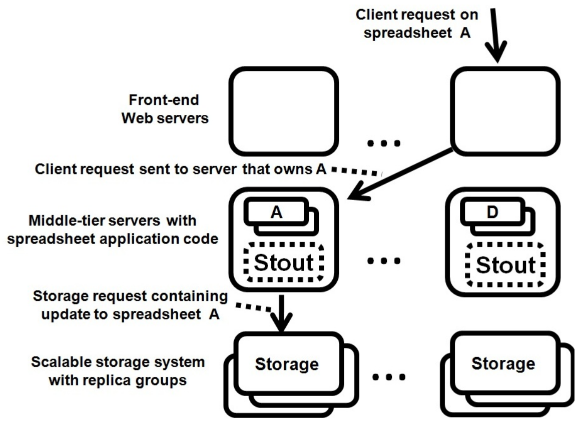

Client request on spreadsheet A
Front-end Web servers
Client request sent to server that owns A
Middle-tier servers with spreadsheet application code
A
Stout
D
Stout
Storage request containing update to spreadsheet A
Scalable storage system with replica groups
Storage
Storage

Figure 1: *Stout in a datacenter spreadsheet application.*

fairness without any explicit coordination across the different application servers utilizing a shared store; and 3) the same adaptation algorithm works well with three different cloud storage systems (a partitioned store that uses Microsoft SQL Server 2008; the PacificA research prototype [26]; and the SQL Data Services cloud store [30]).

2. BACKGROUND

Stout targets *interactive* cloud services. This class of services requires low end-user latency to a variety of data. Stout facilitates high-performance storage access for these services by controlling and adapting the way the services make use of back-end key-value storage systems to provide the best possible response time (i.e., minimize end-user latency). While we believe that Stout's general approach of using a control loop to manage the interactions with a persistent storage tier holds promise for many different kinds of cloud-based services, including those that process large data sets (e.g., services that use MapReduce [10] or Dryad [20]) the rest of this section elaborates on our current target class of interactive latency-sensitive cloud services.

Stout works with scalable services that are *partitioned*. A partitioned service is one that divides up a namespace across a pool of servers, and assigns "keys" within that namespace to only one server at a given point in time. To enable fast response times, the objects associated with the partition keys are stored in memory by these servers. Stout is responsible for handling all interactions with the back-end persistent storage tier. Figure 1 depicts a typical three-tier cloud service, and where Stout fits within that model. The first tier simply consists of front-end Web servers that route end-user requests to the appropriate middle-tier server; the middle tier contains the application logic glued together with Stout, and the back-end tier is a persistent storage system.

As a concrete example, consider an online spreadsheet application, such as that provided by Google Docs [15]. The user-interface component of the spreadsheet application runs inside the client Web browser. As users perform actions within the spreadsheet, requests are submitted to the cloud infrastructure that hosts the spreadsheet service. User requests arrive at front-end Web servers after traversing a network load balancer, and the front-end server routes the user request to the appropriate middle-tier server which holds a copy of the spreadsheet in memory. Each server in the middle tier holds a large number of spreadsheets, and no spreadsheet is split across servers. Whenever the processing of a user request results in a modification to the spreadsheet, the changes are persisted to a scalable back-end storage system before the response is sent back to the user.

Many of today's Web services are built using the same paradigm as the spreadsheet application above. For example, a service for tracking Web advertising impressions can store many "ad counters" at each middle-tier server. Email, calendar, and other online office applications can also use this partitioning paradigm [15, 19, 29].

Forcing writes to stable storage before responding to the user ensures strong consistency across failures in the middle tier; that is, once the user has received a response to her request to commit changes, she can rest assured they will always be reflected by subsequent reads. So long as a middle-tier server maintains these semantics, it is free to optimize the interactions with the storage layer. Thus, when a middle-tier is handling multiple changes, it can batch them together for the storage layer.

3. ADAPTIVE BATCHING

Batching storage requests together before sending them to the store leads to several optimization opportunities (Section 3.1). However, delaying requests to send in a batch is only needed when the store would otherwise be overloaded; if the store is lightly loaded, delaying requests yields a net penalty to client-visible latency. This motivates Stout's adaptation algorithm, which measures current store performance to determine the correct amount of batching as workloads change (Section 3.2).

3.1 Overlapped Request Processing

Having multiple storage requests to send in a batch requires the application to overlap its own processing of incoming client requests. Figure 2 illustrates overlapped request processing for both reads and writes. Note that only reads that miss the middle-tier's cache require a request to the store; cache hits are serviced directly at the middle-tier. Initially, the application receives two client requests, "Change 1 on A" and "Change 2 on B". Both of the client requests are processed up to the point that they generate requests for the store. These are then sent in a single batch to the store. After the store acknowledgment arrives, replies are sent to both of the client requests. While waiting for the store acknowledgment, client re-

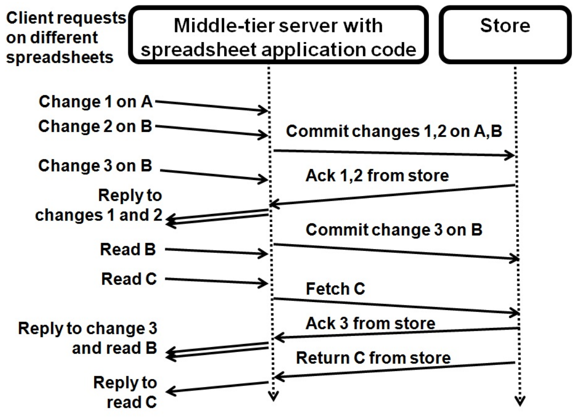


Figure 2: *An example of overlapped request processing.*

quest "Change 3 on B" arrives and is processed up to the point of generating a request to the store. Later, client request "Read B" arrives and hits the middle-tier cache, while "Read C" arrives and requires fetching C from the store. We describe in Section 4 how the Stout implementation handles the multiplexing of these storage requests into batches and the corresponding de-multiplexing of store responses. Grouping storage requests together enables two well-known optimizations:

- **Batching:** Many stores perform better when a set of operations is performed as a group, and many systems incorporate a group-commit optimization [6, 16, 17]. The performance improvements arise from a number of factors, such as reducing the number of commit operations performed on the transaction log, or reducing disk seek time by scheduling disk operations over a larger set. Storage system performance further improves by initiating batching from the middle-tier for reasons that include reduced network and protocol processing overheads.
- **Write collapsing:** When multiple writes quickly occur on the same object, it can be significantly more efficient for the middle-tier server to send only the final object state. An example where write collapsing may arise in cloud services is tracking advertising impressions, where many clients may increment a single counter in quick succession and the number of writes can be safely reduced by writing only the final counter value to the store. Many workloads possess opportunities for write collapsing, and many prior systems are designed to exploit these opportunities [36, 40].

Stout's novelty is managing how these optimizations are exploited for a shared remote store based on a multiplicative-increase multiplicative-decrease (MIMD) control loop. It does this by varying a single parameter, the batching interval. At the end of each interval, Stout sends all writes and cache-miss reads to the store. In this way, the batch size is simply all such reads and writes

		Batching Interval	
	No batching	10ms	20ms
Requests/second	11k	13k	17k
Throughput Gain	-	18%	55%

Table 1: *How a service's maximum throughput can increase by exploiting batching.*

EWMA factor	1/16
$thresh$	0.85
$MinRequests$	10
$MinLatencyFrac$	1/2

Table 2: *Parameters to make measurements and comparisons robust to jitter.*

generated in the previous interval, and write collapsing is obtained to the extent that multiple updates to the same key happened during this interval. Pipelining occurs if this batch is sent to the store while an earlier batch is still outstanding (i.e., when the batching interval is less than the store latency).

For a given workload, a longer batching interval will allow more requests to accumulate, leading to a larger batch size and potentially greater throughput. Table 1 quantifies the improvement in maximum throughput for one of the experimental configurations that we use to evaluate Stout. This configuration is described in detail in Section 5.2. Our goal here is simply to convey the magnitude of potential throughput gain (over 50%) from even slightly lengthening the batching interval. This throughput gain translates into a much larger set of workloads that can be satisfied without queues building up at the store and requests eventually being dropped.

However, the improved throughput of a longer batching interval is not always needed; if the workload is sufficiently light, client latency is minimized by sending every request to the store immediately. For example, the batching intervals that lead to the higher throughput shown in Table 1 also add tens of milliseconds to latency. To determine the right batching interval at any given point in time, Stout measures the current performance of the store. Stout uses these measurements to set its batching interval to be shorter if the store is lightly loaded, and longer if the store is heavily loaded.

3.2 Updating the Batching Interval

The problem of updating the batching interval is a classic congestion control problem: competing requests originate independently from a number of senders (i.e., middle-tiers); these requests have to share a limited resource—the store—and there is some delay before resource oversubscription is noticed by the sender (in this case, the time until the store completes the request). Like TCP, Stout does not require explicit feedback about the degree of store utilization. This allows Stout to be easily deployed with a wide range of existing storage systems. Unlike TCP, Stout must react primarily to delay rather than loss, as stores typically queue extensively before dropping requests. Thus, our design for Stout's control loop borrows from a recent delay-based TCP, Compound TCP (CTCP [37]). In general, delay-based TCP variants react when the current latency deviates from a baseline, falling back to traditional TCP behavior in the event of packet loss. Compared to TCP Vegas [5] (another delay-based TCP), CTCP more rapidly adjusts its congestion window so that it can better exploit high bandwidth-delay product links. For Stout, rapid adjustment means faster convergence to a good batching interval.

However, one aspect of our problem differs from that addressed by congestion control protocols. Delay-based TCP assumes that increasing delay reflects congestion and will consequently reduce the sending rate to alleviate that congestion. Stout acts to reduce congestion by improving per-request performance rather than reducing send rates. Increasing the batch size means that the next request will take longer to process even in the absence of congestion. Furthermore, Stout must distinguish this increased delay due to an increased batch size from increased delay due to congestion. For this reason, Stout has to depart from CTCP by incorporating throughput, not just delay, into measuring current store performance and assessing whether the store is congested.

The remainder of this section describes Stout's approach to updating the batching interval, which we denote by $intrvl$, the time in milliseconds between sending batches of requests to the store. In Section 3.2.1, we describe how Stout decides when it is time to update the batching interval. In Section 3.2.2, we describe how Stout decides whether to increase or decrease the batching interval. Increasing the batching interval corresponds to backing off—going slower because of the threat of congestion—while decreasing the batching interval corresponds to accelerating. Then in Section 3.2.3, we describe how Stout decides how much to increase or decrease the batching interval.

3.2.1 When to Back-off or Accelerate

Like TCP and its many variants, Stout is self clocking: it decides whether or not to back-off more frequently when the store is fast, and less frequently when the store is slow. To this end, Stout tracks the latency between when it sends a request to the store and when it receives a response. Stout computes the mean of these latencies over every request that completes since the last decision to adjust $intrvl$; we abbreviate the mean latency as lat.

Stout decides to either back-off or accelerate as soon as both $MinRequests$ requests have completed and

$(MinLatencyFrac \times lat)$ time has elapsed; the former term is dominant when there is little pipelining, and the latter term is dominant when there is significant request pipelining. We find that this waiting policy mitigates much of the jitter in latency measurements across individual store operations. Table 2 shows the settings for these parameters that we used in our experiments, as well as the other parameters (introduced later in this section) that play a role in making Stout robust to jitter.

3.2.2 *Whether to Back-off or Accelerate*

Stout makes its decision on whether to back-off or accelerate by comparing the current performance of the store to the performance of the store in the recent past. We denote the store's current performance by $perf$, its recent performance by $perf^*$, and we explain how both are calculated over the next several paragraphs. As mentioned in the Introduction, Stout restricts its measurements to response times so that it can be re-used on different stores, as this measurement requires no store-specific support. The performance comparison is done with some slack (denoted as $thresh$), so as to avoid sensitivity to small amounts of jitter in the measurements:

```
if ($perf < (thresh \times perf^*)$)
    BACK-OFF
else
    ACCELERATE
```

We calculate $perf$ using the number of bytes sent to and received from the store during the most recent self-clocking window (denoted by $bytes$), the mean latency of operations that completed during this same period of time, and the length of the current batching interval. (Note that higher $perf$ is better.)

$$perf = \frac{bytes}{lat + intrvl}$$

Our $perf$ definition is a simple combination of latency and throughput: Stout's latency is $intrvl + lat$, the time until Stout initiates a batch plus the time until the store responds; Stout's throughput is $bytes/intrvl$, the amount sent divided by how often it is sent.

Incorporating throughput appropriately rewards backing-off when it causes throughput to increase and the throughput improvement outweighs the larger store latency (lat) from processing a larger batch. By contrast, just measuring latency could lead to an undesirable feedback loop: Stout could back-off (taking more time between batches), each batch could send more work and hence take longer, the store would appear to be performing worse, and Stout could back-off again.

Stout must compute recent performance ($perf^*$) in a manner that is robust to background noise, is sensitive to the effects of Stout's own decisions, and that copes with delay between its changes and the measurement of those changes. To this end, Stout computes $perf^*$ over different sets of recent measurements depending on its own recent actions (e.g., backing off or accelerating). To explain the $perf^*$ computation, we first present the algorithm and then provide its justification.

```
if (last decision was ACCELERATE)
    $perf^* = \text{MAX}_i(\frac{bytes_i}{lat_i + intrvl_i})$                        (1)
else // last decision was BACK-OFF
    if ($intrvl < \text{EWMA}(intrvl_i)$)
        $perf^* = \frac{\text{EWMA}(bytes_i)}{\text{EWMA}(lat_i) + \text{EWMA}(intrvl_i)}$    (2)
    else // ($intrvl \geq \text{EWMA}(intrvl_i)$)
        $perf^* = \frac{\text{EWMA}(bytes_i)}{\text{EWMA}(lat_i) + intrvl}$              (3)
```

Equation (2) for computing $perf^*$ is the most straightforward: it is an exponentially weighted moving average (EWMA) over all intervals i since the last acceleration. However, Stout cannot always wait for latency changes to be reflected in this EWMA because of the risk of overshooting—not reacting quickly enough to latency changes that Stout itself is causing. This risk motivates Equations (1) and (3), which we now discuss.

Equation (1) prevents overshoot while accelerating. When Stout is accelerating, it runs a risk of causing the store to start queuing. To prevent this, Stout heightens its sensitivity to the onset of queuing by computing recent performance ($perf^*$) as the best performance since the last time Stout backed-off. Stout stops accelerating as soon as current performance drops behind this best performance. By contrast, calculating recent performance using an EWMA would mask any latency increase due to queuing until it had been incorporated into the EWMA multiple times.

Equation (3) prevents overshoot while backing-off: when Stout backs off, the increase in $intrvl$ can penalize current $perf$, potentially causing Stout to back-off yet again, even if throughput (the $bytes/lat$ portion of $perf$) has improved. To address this, when the current $intrvl$ is larger than its recent history, we use it in calculating both $perf$ and $perf^*$.

3.2.3 *How Much to Back-off or Accelerate*

Stout reuses the MIMD-variant from CTCP [37]: MIMD allows ramping up and down quickly, and as in CTCP, incorporating $\sqrt{intrvl}$ into the update rule provides fairness between competing clients. A minor difference between CTCP and Stout is that CTCP modifies the TCP window, and backing-off corresponds to decreasing this window; Stout modifies its batch interval, and backing-off corresponds to increasing this interval.

Stout backs off using a simple multiplicative back-off step, and it accelerates using a multiplicative factor that decreases as $intrvl$ approaches its lower limit (1 ms in

α'	1/400
α_{max}	1/2
β	1/10
$intrvl_{initial}$	80 ms
$intrvl_{max}$	400 ms

Table 3: *Parameters for gain and boundary conditions. These parameters are analogous to those in CTCP, e.g., $intrvl_{max}$ corresponds to RTO_{max}.*

this case):

BACK-OFF:

$$intrvl_{i+1} = (1 + \alpha) * intrvl_i$$

ACCELERATE:

$$intrvl_{i+1} = (1 - \beta) * intrvl_i + \beta * \sqrt{intrvl_i}$$

Competing clients converge to fairness because slow clients accelerate more than fast clients when the store is free, and all clients back-off by an equal factor when the store is busy. The CTCP paper formally analyzes this convergence behavior [37].

The incremental benefit of additional batching decreases as the batch size grows. Because of this, Stout must react more dramatically if the store is already processing large batches and then starts to queue. To accomplish this, we make α (the back-off factor) proportional to an EWMA of latency, with an upper bound:

$$\alpha = \text{MAX}(\text{EWMA}(lat_i) * \alpha', \alpha_{max})$$

Finally, stores occasionally exhibit brief pauses in processing, leading to short-lived latency spikes (this behavior is described in greater detail in Section 5.7). This behavior could cause Stout to back-off dramatically, and then take a long time recovering. To address this, we introduce an $intrvl_{max}$ parameter; just as TCP will never assume that the network has gotten so slow that retransmissions should wait longer than RTO_{max}, Stout will never assume that store performance has degraded to the point that batches should wait longer than $intrvl_{max}$. This bounds Stout's operating range, but allows it to recover much more quickly from brief store pauses.

Table 3 shows the gain and boundary condition parameter settings. As in CTCP's parameter settings, the initial batching interval is conservative, and the gain parameters lead to bigger back-offs than accelerations, similar to how TCP backs off faster than it accelerates. Our experiments in Section 5 show that Stout works well with these choices, and that it effectively converges to batching intervals spanning over two orders of magnitude.

4. IMPLEMENTATION

Stout's primary novelty is its algorithm for dynamically adjusting the batching of storage requests. We implement the Stout prototype to evaluate this algorithm

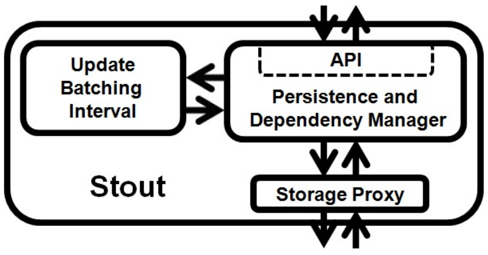
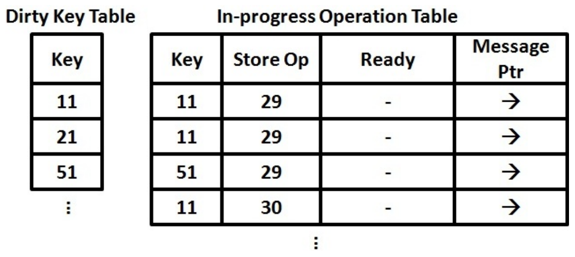


Figure 3: *The internal architecture of Stout.*



Dirty Key Table

Key
11
21
51
⋮

In-progress Operation Table

Key	Store Op	Ready	Message Ptr
11	29	-	→
11	29	-	→
51	29	-	→
11	30	-	→
⋮			

Figure 4: *Data structures for Stout's dependency map.*

with a real-world cloud service (a component of Microsoft's Live Mesh service [27]). We first describe how the application ensures that each key is owned by a single middle-tier (Section 4.1). We then describe the Stout internal architecture (Section 4.2), followed by how Stout multiplexes storage requests into batches and the corresponding de-multiplexing of store responses (Section 4.3). Finally, we describe the Stout API by walking through an example of its use (Section 4.4).

4.1 Key Ownership

As discussed previously, applications that use Stout must ensure that all requests on a given partition key are handled by only one middle-tier server at any given point in time. In particular, the write collapsing optimization requires that all updates to a given partition key are being sent to the same server. This requirement could be met using a variety of techniques; the applications we evaluate rely on Centrifuge [2].

Centrifuge is a system that combines lease-management with partitioning. Centrifuge uses a logically centralized manager to divide up a flat namespace of keys across the middle-tier servers. Centrifuge grants leases to the middle-tiers to ensure that responsibility for individual objects within the namespace are assigned to only one server at any given point in time. Front-end Web servers route requests to middle-tiers via Centrifuge's lookup mechanism.

4.2 Stout Internal Architecture

Stout's internal architecture divides the problem of managing interaction with the store into three parts, as depicted in Figure 3. The "Persistence and Dependency

Stout.Fill(key)	Ask Stout to fetch objects associated with partition key from store.
Stout.MarkDirty(key)	Mark objects associated with partition key as modified, so that Stout knows to persist them.
Stout.MarkDeleted(key)	Mark objects associated with partition key as deleted, so that Stout knows to delete them from store.
Stout.SendMessageWhenSafe(key, sendMsgCallback)	Sends a reply message after Stout's internal dependency map indicates it is safe to send response.
Stout.SerializeDone(key[], byte[][])	App indicates completion of Stout's request to serialize objects.
App.Serialize(key[])	Callback invoked by Stout for objects that have been marked dirty. Requests App to convert objects into byte arrays to send to the store and respond with *SerializeDone()*.
App.Deserialize(key[], byte[][])	Callback invoked by Stout when *Fill()* responses arrive from store. Converts each byte[] into object.

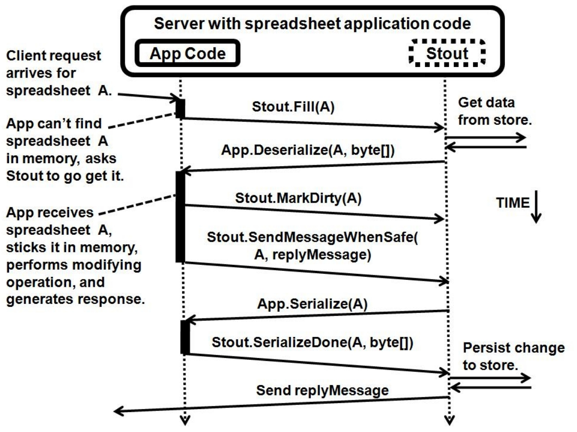
Table 4: *Client API. All calls are asynchronous.*

```
// We have received a message containing ``update''
// for the spreadsheet named by ``key''.
ProcessRequest(update, key) {
  // If we don't have the state for this key,
  // we ask Stout to get it from the back-end store.
  if (table[key] == null)
    Stout.Fill(key);
  ... // Block until Stout has filled in table[key].

  // Spreadsheet-specific logic is in UpdateSheet().
  replyMessage = UpdateSheet(table[key], update);
  Stout.MarkDirty(key); // Tell Stout about update.

  // Ask Stout to send reply when update is persisted.
  Stout.SendMessageWhenSafe(key, replyMessage);
}
```

(a) *Placement of API calls in sample application code. Stout and the application communicate via message passing, so the application does not need to coordinate its locking with Stout.*


Server with spreadsheet application code
App Code
Stout
Client request arrives for spreadsheet A.
App can't find spreadsheet A in memory, asks Stout to go get it.
Stout.Fill(A)
Get data from store.
App.Deserialize(A, byte[])
TIME
App receives spreadsheet A, sticks it in memory, performs modifying operation, and generates response.
Stout.MarkDirty(A)
Stout.SendMessageWhenSafe(A, replyMessage)
App.Serialize(A)
Stout.SerializeDone(A, byte[])
Persist change to store.
Send replyMessage

(b) *Flow of calls between spreadsheet application, Stout and store. The portion of time when the spreadsheet application is active is denoted by the thick black line.*

Figure 5: *An example use case of a spreadsheet application interacting with Stout.*

Manager" component handles correctness and ordering constraints (e.g., ensuring that requests are committed to the store before replies are sent), as described in Section 4.3. Applications interact with this component through the API described in Section 4.4. The "Update Batching Interval" component implements the adaptive batching algorithm from Section 3.2. The "Storage Proxy" component is a thin layer that connects Stout to a specific scalable storage system. We have implemented three proxies to interface Stout with different cloud storage systems, and all three use TCP as a transport layer.

4.3 Persistence and Dependencies

Each middle-tier uses Stout to manage its in-memory data as a coherent cache of the store. Stout is responsible for communicating with the store and ensuring proper message ordering. The application is then responsible for calling Stout when it: (1) needs to fetch data from the store, (2) modifies data associated with a partition key, or (3) wants to send a reply to a client.

Stout ensures proper message ordering by maintaining a dependency map that consists of two tables, as depicted in Figure 4. Keys are added to the table of dirty keys whenever the application notifies Stout that a key has been modified. Messages provided by the application are added to the table of in-progress operations if the key is dirty or there are any outstanding operations to the store on this key; otherwise, the messages are sent out immediately. When Stout sends a batch of writes to the store to commit the new values of some keys, those same keys are removed from the table of dirty keys, and Stout fills in the "Store Op" for the appropriate rows in the table of in-progress operations. When a store operation returns, Stout sends out messages in the order they were received from the application.

Figure 4 depicts both batching (keys 11 and 51 were both sent in storage operation 29) and write collapsing (two update operations for key 11 were both conveyed in operation 29). Stout requires the store to commit operations in order, but the store may still return acknowledgments out of order. In our example, if the acknowledgment of 30 arrives before the acknowledgment of 29, Stout would mark the fourth row of the table "Ready" and send the message once all earlier store operations on key 11 are ready and their messages sent.

4.4 Stout API

Table 4 describes each of the API calls and the callbacks that applications must provide for Stout. Figure 5a shows how a datacenter spreadsheet application places the API calls in its code. Before the application's *ProcessRequest()* function is called, the application has already received the request, done any necessary authentication, and checked that it holds the lease for the given partition key. *ProcessRequest()* handles both modifying spreadsheet objects (done in *UpdateSheet()*) and interacting with Stout: using Stout to fetch state from the store, letting Stout know that the state has been updated, and telling Stout about a reply that should be sent once the update has been persisted to the store. We do not show the code to send the reply, but note that before the application sends the reply message to the client, it must check that the lease for the partition key has been continuously held for the duration of the operation.

Figure 5b illustrates the ordering of calls between the application and Stout, and between Stout and the store. When an application or service first receives a request on a given partition key, it fetches the state associated with that partition key using the *Stout.Fill()* call. When the state arrives, Stout calls *App.Deserialize()* to create in-memory versions of fetched objects, which can then easily be operated on by the application logic.

To support coherence, Stout needs to know when operations modify internal service state, so that these updates can be saved to the store. Since Stout has no *a priori* knowledge of the application internals, Stout requires the service developer to call *Stout.MarkDirty()* in any service methods that modify objects associated with a partition key. At some point after a key has been marked as dirty, the Stout persistence manager will call *App.Serialize()* on a set of dirty keys. By delaying calls to *App.Serialize()*, Stout allows modifications to the same object to overwrite each other in-memory, thus capturing write collapsing. The application then responds by calling *Stout.SerializeDone()* with the corresponding byte arrays to be sent to the store.

When a Stout-enabled service would like to send a response to a user's request, it must use *Stout.SendMessageWhenSafe()* to provide the outgoing message callback to Stout. Stout will then take responsibility for determining when it is safe to send the outgoing message, based on its knowledge of the current interactions with the persistent store related to the partition key for that request. For example, if the message is dependent on state which has not yet been committed to the persistent store, Stout cannot release the message until it receives a store acknowledgment that the commit was successful.

For certain services, the state associated with a partition key may be large enough that one does not want to serialize the entire object every time it is modified, especially if the size of the modifications is small compared to the size of the entire state. To handle this case, the API supports an additional parameter, a sub-key. Stout keeps track of the set of dirty sub-keys associated with each partition key, and asks the application for only the byte arrays corresponding to these sub-keys. Finally, Stout also enables deletion from the persistent store using the *Stout.MarkDeleted()* call, which similarly takes both partition keys and sub-keys. Stout tracks these requested deletes, and then includes them in the next batch sent to the store, along with any read and write operations.

5. EVALUATION

We now demonstrate the benefits of Stout's adaptation strategy. In Section 5.1, we describe the setup for our experiments. In Section 5.2, we evaluate the potential benefits of batching and write collapsing in the absence of adaptation. In Sections 5.3-5.6, we evaluate Stout's adaptation strategy and show that it outperforms fixed strategies with both constant and changing workloads, that multiple instances of Stout dynamically converge to fairly sharing a common store, and that Stout's adaptation algorithm works across three different cloud storage systems. Finally, in Section 5.7, we examine the behavior of our store, and we show that Stout is robust to brief "hiccups" where the store stops processing requests.

5.1 Experimental Setup

We first describe the application that we ported to use Stout and this application's workload, and we then characterize the system configuration for our experiments.

5.1.1 Application and Workload

The application we run on our middle-tier servers is a "sectioned document" service. This service is currently in production use, and additional details can be found in the Centrifuge paper [2]. This service allows documents to contain independent sections that can be named, queried, added, and removed. The unmodified service is approximately 7k commented lines of C# code, and we ported this service to use the Stout API changing approximately 300 lines of code. Stout itself consists of 4k commented lines of code and the storage proxies are each approximately 600 commented lines of code.

In production, this service is deployed on multiple large pools of machines. One pool is used exclusively to store device presence: a small amount of addressing information, such as IP address, and an indication whether the device is online. Although we were unable to obtain a trace from production, we used known characteristics of the production system to guide the design of a synthetic client workload for our evaluation: varying request rates on a large number of small documents, 2k documents per middle-tier, each consisting of a single 256-byte section. At saturation, our store is limited by the total number of operations rather than the total number of bytes being stored under this workload, a common situation [7, 31].

In this synthetic workload, we designed the read/write mixture to best evaluate Stout's ability to adapt under workload variation. We avoid making the workload dominated by reads, because this would have primarily loaded the middle-tiers, and Stout's goal is to appropriately adapt when the store is highly loaded. We also avoided a pure-write workload because this would not capture how reads that hit the middle-tier cache are delayed if they touch documents that have been updated but where the update has not yet been committed to the store. This led us to choose a balanced request mixture of 50% reads and 50% writes.

In the commercial cloud service that motivates our workload, all data fits in RAM—Stout is using the store for persistence, not capacity. Because of this, read latencies are uniformly lower than write latencies (e.g., Figure 9 in Section 5.3). In the Stout consistency model, write latencies impact the user experience because responses are only sent after persisting state changes (e.g., after saving a spreadsheet update). Because writes form the half of the workload that poses the greater risk of poor responsiveness, the rest of the evaluation reports only write latencies unless otherwise noted.

5.1.2 System Configuration

Our testbed consists of 50 machines with dual-socket quad-core Intel Xeon 5420 CPUs clocked at 2.5 GHz, with 16 GB of RAM and 2×1 TB SATA 7200 rpm drives. We chose the ratio of front-ends to middle-tiers to storage nodes such that the overall system throughput was maximized subject to the constraint that the storage system was the bottleneck. This led to dividing the 50 machines into 1 experiment controller, 1 Centrifuge lease manager, 12 front-ends that also generate the synthetic client workload, 32 middle-tiers using the Stout library, and 4 systems running the persistent storage system. The choice of 32 middle-tiers means there are 64k total documents in the system. Unless noted otherwise, latency is measured from the front-ends (denoted FE latency in the figures)—this represents the part of end-to-end client latency due to the datacenter application.

Most of our experiments run Microsoft SQL Server 2008 Express on each of the four storage servers to implement persistent storage. We configure the storage servers to use a dedicated disk for SQL logging, and we followed the SQL documentation to ensure persistence under power loss, including disabling write-caching on our SATA drives [12]. The Stout storage proxy consists of a simple client library that performs hash-based partitioning of the database namespace. For a small number of experiments, we used two additional stores: the PacificA storage system [26] which uses log-based storage and replication, and the commercially available SQL Data Services (SDS) cloud-based storage system [30].

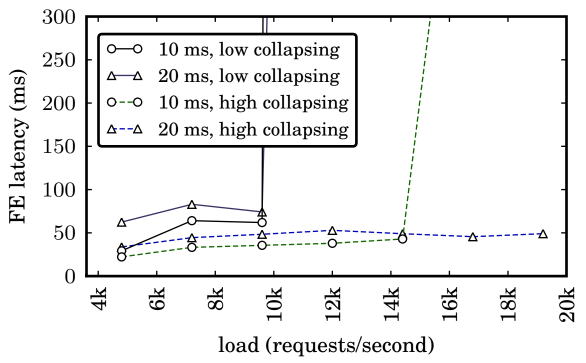


Figure 6: *Two fixed batching intervals (10 ms, 20 ms) on a workload with low write collapsing (10k documents) or high write collapsing (100 documents).*

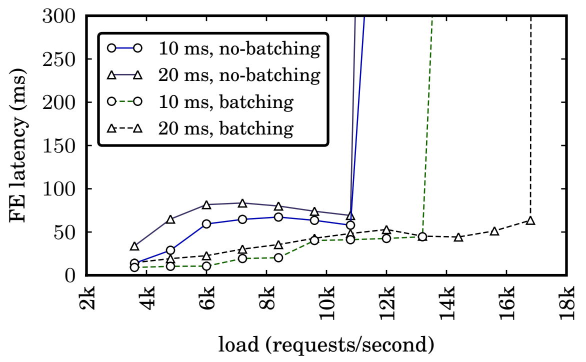


Figure 7: *Two fixed batching intervals (10 ms, 20 ms) on identical workloads with and without batching.*

Under our workload, these stores occasionally exhibit brief hiccups where they pause in processing; we describe this in more detail in Section 5.7. Unless noted otherwise, we report data from runs without hiccups.

5.2 Batching and Write Collapsing

We perform two experiments to evaluate the potential performance improvements that are enabled by the batching and write collapsing optimizations. For both experiments, we use two different fixed batching intervals—10 and 20 ms—to isolate the benefits of batching and write collapsing from adaptation.

Figure 6 shows the performance benefits of write collapsing. For this experiment, requests are delayed for the duration of the batching interval, but they are not actually sent in a batch; at the end of each batching interval, all the accumulated requests are sent individually to the store. Because of this, the entire observed performance difference is due to write collapsing. The low collapsing workload consists of 10k documents spread across the 32 middle-tiers, while high collapsing consists of only 100 documents, significantly increasing the probability that there are multiple updates to the same document within the batching interval. The graph shows that, as expected, write collapsing reduces latency and improves

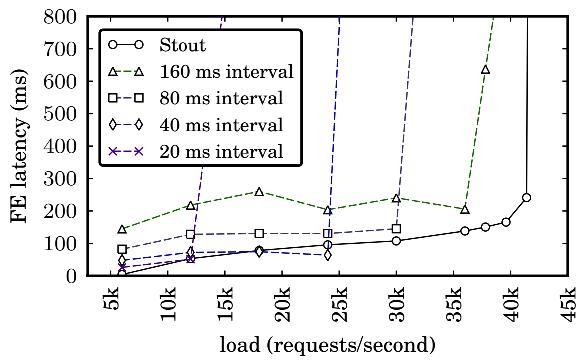


Figure 8: *Mean response latency for writes: Stout versus fixed batching intervals over a wide variety of loads.*

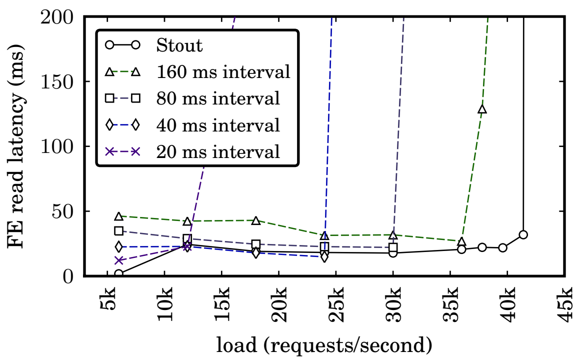


Figure 9: *Mean response latency for reads: Stout versus fixed batching intervals over a wide variety of loads. Note that the y-axis is 4× smaller than in Figure 8.*

the capacity of the system. For the low collapsing case, we see that the 10- and 20-ms batching intervals can satisfy between 4k requests/second and 10k requests/second with better client perceived latency for a 10-ms batching interval. However, at 12k requests/second the storage system is overloaded, resulting in a large queuing delay represented by an almost vertical line. In contrast, for the high collapsing workload a 10-ms batching interval can sustain nearly 15k requests/second because the actual number of writes sent to the store is reduced. For the 20-ms batching interval, the number of writes is reduced enough to shift the bottleneck from the store to the middle-tier and provide up to 80k requests/second.

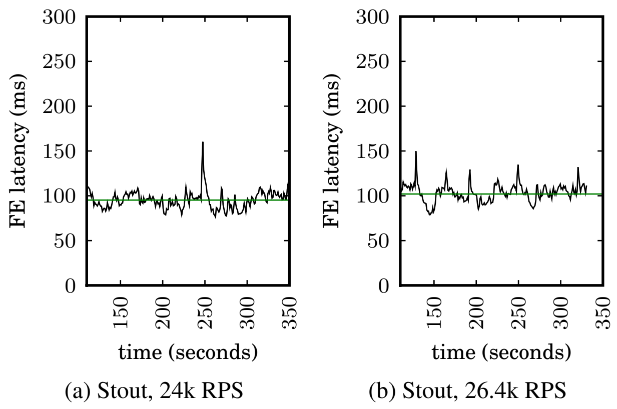
Figure 7 shows the performance benefits of batching. The no-batching experiments reflect disabling batching using the same methodology as in the write collapsing experiment: requests are delayed but then sent individually. We see that the throughput benefits of batching are noticeable at 10 ms, and they increase as the batching interval gets longer, which in turn causes the batch size to get larger. At a 20-ms batching interval, batching allows the system to handle an additional 6k requests per second. The amount of write collapsing for each fixed batching interval in this experiment is constant (and small). We separately observed that PacificA also delivers performance benefits from batching (this is detailed in Section 5.6, where we evaluate Stout on both PacificA and SDS). As mentioned in the Introduction, the reason for batching's benefits depend on the individual store being used; for our partitioned store built on SQL, we separately determined that a significant portion of the benefit comes from submitting many updates as part of a single transaction.

5.3 Adaptive vs. Fixed Batching

In this section, we demonstrate that Stout is effective across a wide operating range of offered loads, and investigate the overhead imposed by Stout's adaptation over the *best* fixed batching interval at a given load.

Figures 8 and 9 compare Stout to fixed batching intervals that vary from 20 ms up to 160 ms, for offered loads that range from 5k requests/second all the way up to 41k requests/second, which is very near the maximum load that our storage system can support. These figures are generated from the same experiments: Figure 8 shows the mean response latency for write operations whereas Figure 9 shows the latency for reads – all reads are cache hits in this workload, but the latency numbers do include delay from reading an updated document where the update has not yet been committed to the store. In both graphs, we see that Stout provides a wider operating range than any of the fixed batching intervals, and it provides response latencies that are either similar to or better than the fixed batching intervals. Looking at the two extremes of latency and throughput in Figure 8, Stout's 4.2 ms latency at 6k requests/second is over 34× smaller than the 144 ms latency incurred by the longest fixed batching interval in this experiment (160 ms), while Stout's 41k requests/second maximum is over 3× larger than the 12k requests/second maximum for the shortest fixed batching interval in this experiment (20 ms).

To understand the overhead of Stout's adaptation, we compare Stout to different fixed batching intervals at fine granularity under two fixed workloads. In Figures 10 (a) and (b), the time series show Stout's latency to be relatively steady, and for this reason we focus on the mean latency throughout this section. Figure 10 (c) compares Stout's mean to fixed intervals with an offered load of 24k requests/second. The best fixed interval is at 50 ms, and here we observe that Stout's adaptation adds just under 15 ms to the response latency (from 80 to 94 ms) and is within the standard deviation. When the fixed batching interval is too short (40 ms), the store is overloaded and we see large queuing delays. When the fixed interval is too long (at 70 ms and above), we see unnecessary latency. Figure 10 (d) shows a similar comparison, but with an offered load of 26.4k requests/second. Here we see



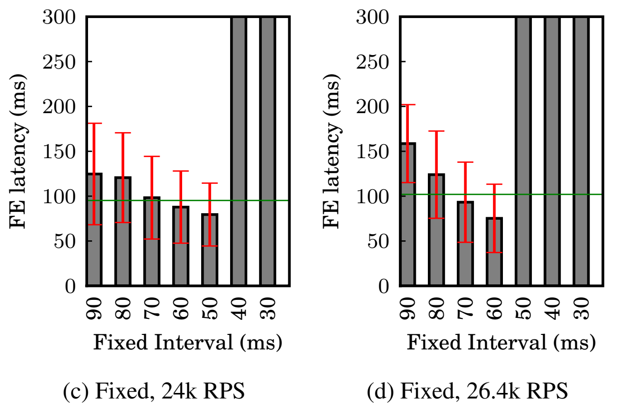


Figure 10: *Latency of responses for Stout (a, b) and fixed batching intervals (c, d), at two different workloads, 24k requests/second (a, c) and 26.4k requests/second (b, d). In (a, b), we see Stout's changing response latency overlayed with its mean response latency. In (c, d), Stout's mean response latency is overlayed with the mean latency and standard deviation for multiple fixed batching intervals. The slight increase in requests/second causes the best fixed interval from 24k requests/second to generate queuing at 26.4k requests/second.*

that the best fixed interval is at 60 ms, and the overhead imposed by Stout's adaptation is about 25 ms (from 75 to 100 ms), again within the standard deviation. If we use the best fixed interval from 24k requests/second (50 ms), the store becomes overloaded and unable to process requests in a timely fashion until the load subsides. These results demonstrate the need for adaptation—choosing the right fixed interval is difficult, even with this modest difference in offered load.

5.4 Dynamic Load Changes

Thus far we have shown Stout operating over fixed request rates. Here, we explore Stout's response to a sudden change in request load. For this experiment we apply a fixed load of 12k requests/second to our standard configuration and part way through the experiment we change the request load. Figure 11 shows the front-end latency for two of these experiments. In the first experiment, the load decreases to 6k requests/second. The front-end latency for Stout decreases from 50 ms to 5 ms. In the second experiment, the load increases to 18k requests/second and the latency increases from 50 ms to 80 ms. In contrast, a 20-ms fixed interval is marginally better than Stout at 12k requests/second but it only achieves 24 ms after the decrease and it causes queuing at the store after the increase. This demonstrates Stout's benefits in the presence of workload changes.

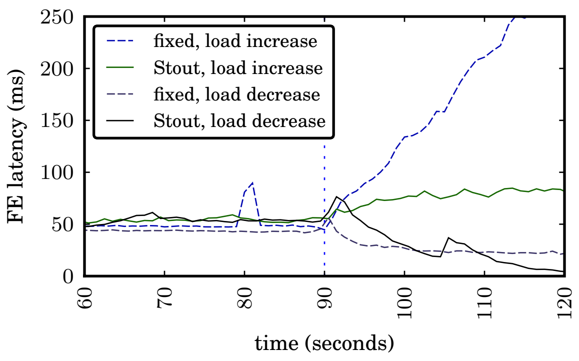


Figure 11: *Stout outperforms a fixed batching interval after the load either increases or decreases.*

5.5 Fairness

Cloud storage systems typically serve many middle-tiers and it is important that these middle-tiers obtain fair usage of the store. To measure Stout's ability to converge to fairness, we ran an experiment where after 90 seconds, we forcibly set half of the thirty-two middle-tiers to a batching interval of 400 ms and the remaining half to 80 ms. The middle-tier servers then collectively reconverge to the steady state. Because Centrifuge balances the distribution of documents across the middle-tiers, they have identical throughput throughout the experiment and we are only concerned with latency-fairness. The middle-tiers achieve good fairness after re-convergence: measuring from 30 seconds after the perturbation to 120 seconds after the perturbation, the mean latencies have a Jain's Fairness [22] of 0.97, where a value of 1.0 is optimal.

5.6 Alternate Storage Layers

To explore the generality of Stout's adaptation algorithm, we run experiments using two additional storage platforms with substantially different architectures. For both, we keep the same algorithm but calibrate the parameters to the new store. We first evaluate Stout against SQL Data Services (SDS) [30], a pre-release commercial storage system. For SDS, we calibrate the parameters to be the same as in Section 3.2 except that $thresh = 0.2$

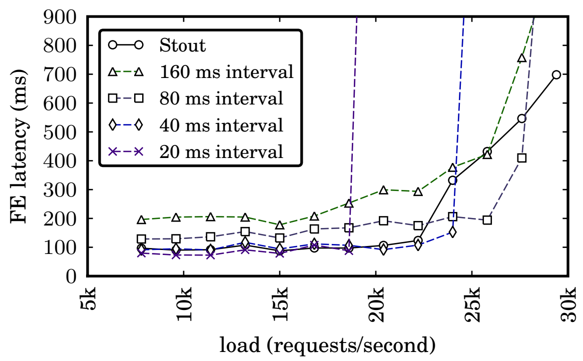


Figure 12: *Mean response latency for writes using PacificA: Stout and fixed intervals over a variety of loads.*

and $\beta = 1/4$. The current SDS API does not support batching or pipelining, and thus the best approach in our workloads is to send as rapidly as possible. We find that Stout does converge to sending as rapidly as possible.

We also evaluate Stout against PacificA [26], a research system that differs from our SQL-based storage layer in that it includes replication and uses log-based storage. We configure PacificA with three-way partitioning and three-way replication for a total of nine storage machines and one additional metadata server. The rest of the setup consists of twelve front-ends, sixteen middle-tiers, and one Centrifuge manager server. We calibrate the parameters from Section 3.2 to have EWMA-factor= $1/32$, $thresh$ = 0.7, and $\beta = 1/8$. Figure 12 shows Stout's behavior across a range of request loads. At low to moderate load, Stout compares favorably to the best (20- and 40-ms) fixed batching intervals. As load increases, PacificA's log compaction frequency also increases, resulting in sufficiently frequent store hiccups that we are not able to avoid them in our experiments. After 22.2k requests/second, Stout has difficulty differentiating the store hiccups from the queuing behavior to which it is adapting. In spite of these hiccups, Stout outperforms any fixed batching interval in the presence of significant workload variation: compared to the short intervals, it avoids queuing at high loads; compared to long intervals, it yields much better latency at low loads.

5.7 Store Hiccups

As mentioned in our experiments with PacificA, stores sometimes experience hiccups, where they briefly pause in processing new requests. Such Stout-independent hiccups can lead to large spikes in observed latency, complicating Stout's task of inferring store load. We now investigate the issue of hiccups in more detail.

Figure 13 shows the occasional brief pauses in processing (or "hiccups") that occur over a 2-hour interval when using the SQL Server storage system. For this experiment, we used a single middle-tier server sending 3k operations per second with a fixed 2-ms batching interval to a single SQL Server back-end machine, and we

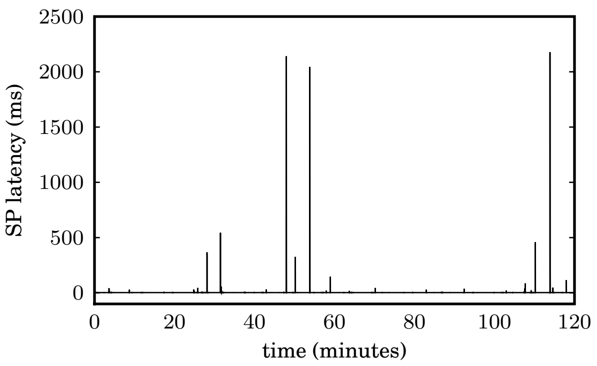


Figure 13: *Intermittent hiccups in store processing yield brief spikes in latency as measured from the middle tier. These measurements were taken with a 2-ms fixed batching interval and 3k requests/second.*

measured latency from within the Stout storage proxy — this is denoted SP latency and it only includes the time to send the requests over a TCP connection to the back-end and the time that the store takes to service these requests and send responses back to the middle-tier. The figure shows that these hiccups occur on an irregular and infrequent basis, and they lead to significant spikes in latency — up to three orders of magnitude greater than the steady state. Although this figure only shows the hiccups at one offered load, we have run similar experiments with different loads, and we have not observed any obvious correlation between the offered load and the frequency of hiccups in this store.

Although we do not know the exact cause of hiccups in the SQL store, we believe they are caused by periodic background bookkeeping tasks that are common in storage systems. We did make efforts to eliminate such hiccups from SQL Server by both disabling the option that generates query-planning statistics and setting the recovery interval to one hour (the recovery interval controls how much replay from the log may be needed after a crash). These changes reduced the number of hiccups but did not eliminate them. As mentioned in Section 5.6, we observed that log compaction is responsible for even more frequent hiccups in PacificA.

Because these brief latency spikes may be unrelated to the offered load, an appropriate response to them is simply to pause briefly; increasing the batching interval is not appropriate because the store is not actually overloaded. The problem of a unrelated event causing the appearance of congestion is familiar from the literature on TCP over wireless channels, where packet loss may reflect either congestion (which should be mitigated by the sender) or background channel noise (which can frequently be ignored). In response, researchers have proposed explicit signaling techniques like ECN [4, 25] to improve performance in these challenging environments. Our measurements suggest that similar mechanisms for

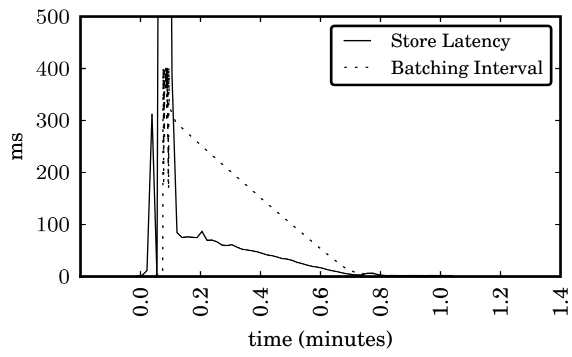


Figure 14: *Stout recovering from a store hiccup while operating at 3k requests/second.*

adaptive use of cloud storage are also worth researching. In this paper, we restrict our attention to showing that Stout, which does not try to distinguish latency due to store hiccups from latency due to overload, still copes acceptably with such hiccups.

Figure 14 shows how Stout reacts to one of these hiccups: the solid line shows the measured response time of the store, and the dashed line shows how Stout adjusts its $intrvl$ as a result of the latency spike. With $intrvl_{max}$ set to 400 ms, Stout takes slightly over half a minute to recover from the very large spike in latency (the peak in this figure is 2,696 ms) caused by this hiccup. This recovery is rapid compared to the frequency of hiccups. Lowering $intrvl_{max}$would improve recovery time, but would also reduce Stout's operating range.

The rarity of store hiccups raises a methodological question: each of our experiments would have needed to run for hours in order for the number of hiccups to be similar across runs. Because Figure 8 alone includes 27 such experiments, such an approach would have significantly hindered our ability to evaluate Stout under a wide variety of conditions. Because Stout recovers from store hiccups with reasonable speed, we chose instead to re-run the occasional experiment that saw such a hiccup. The one exception is our experiment using PacificA (Section 5.6), where hiccups were sufficiently frequent that we did not need to take any special steps to ensure a comparable number across runs.

6. RELATED WORK

Stout's control loop is inspired by the literature on TCP and, more generally, adaptive control in computer systems. The Stout implementation also incorporates a number of well-known techniques from storage systems. We briefly discuss a representative set of this related work.

There is a large existing literature on TCP [21, 24, 43]. This prior work has explored many different indicators of utilization and load; Stout uses response time measurements to adjust its rate of sending requests to the store. In this regard, Stout is similar to TCP Vegas [5], FAST TCP [41] and Compound TCP (CTCP) [37], each of which attempts to tune the transmit rate of a TCP flow based upon the inter-packet delay intervals. In comparison, Stout's control loop has to deal with the additional subtlety of distinguishing delay due to congestion from delay due to sending a larger batch.

Control theory is a deep field with many applications to computer systems [42, 38, 8, 34, 28, 9]. Despite these successes, many adaptation problems in computer systems have remained unaddressable by control theory due to the dramatic differences between computer systems and the systems that control theory has traditionally considered [18]. For example, advocates of a class of controllers called self-tuning regulators have constructed a list of eight requirements that computer systems must satisfy to enable their successful application [23]. Scale-out storage systems fail to satisfy a number of these conditions, such as the requirement for a modest bound on the actuation delay of the system (e.g., if an application enqueues a large number of requests, future request batching can take a very long time to reduce user-perceived latency). Other control techniques may remove this particular requirement, but instead introduce other difficult requirements, such as the need for a detailed model of scale-out storage system performance [23].

The Stout implementation borrows from prior work on storage systems in two major ways. First, the performance benefits of batching, write collapsing and pipelining are well-known, and have been leveraged by systems such as Lightweight Recoverable Virtual Memory (LRVM) [36], Low-Bandwidth File System [32], Farsite [1], Cedar [16], Practical BFT [6], Tandem's B30 system [17] and the buffer cache [40]. Stout's novelty is in using a control loop to manage exploiting these optimizations, not the optimizations themselves.

Second, Stout's internal architecture incorporates at least two major ideas from prior storage systems. Splitting consistency management from storage was explored in Frangipani [39] and LRVM [36], while prior work such as Soft Updates [14], Generalized File System Dependencies [13], and xsyncfs [33] explored ways to provide some or all of the performance benefits of delayed writes with better consistency guarantees.

7. CONCLUSION

Stout's adaptation algorithm is the first technique for automatically adapting application usage of scalable key-value storage systems. Stout treats store access as a congestion control problem, measuring the application-perceived latency and throughput of the store, and dynamically adjusting the application's grouping of requests to the store. To evaluate this algorithm, we implemented the Stout system and modified a real-world cloud service to use Stout. We found that in the presence of significant workload variation, Stout dramatically outperforms non-adaptive approaches.

Acknowledgements

We thank the anonymous reviewers, and our shepherd, Garth Gibson, for their feedback on earlier drafts of this manuscript.

8. REFERENCES

[1] A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, J. Howell, and J. Lorch. Load management in a large-scale decentralized file system. Technical Report MSR-TR-2004-60, Microsoft Research, July 2004.

[2] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. In *Proceedings of USENIX NSDI*, Apr. 2010.

[3] Azure Storage. `http://www.microsoft.com/azure/windowsazure.mspx`.

[4] H. Balakrishnan and R. Katz. Explicit Loss Notification and Wireless Web Performance. In *Proceedings of the IEEE Globecom Internet Mini-Conference*, 1998.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM*, pages 24–35, Aug. 1994.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX OSDI*, 1999.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of USENIX OSDI*, Nov. 2006.

[8] J. S. Chase, D. C. Andersen, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *Proceedings of ACM SOSP*, pages 103–116, Oct. 2001.

[9] C. M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of VLDB*, pages 342–353, Aug. 1993.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of USENIX OSDI*, Dec. 2004.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM SOSP*, pages 205–220, Oct. 2007.

[12] Disable SATA Write Caching. `http://support.microsoft.com/kb/811392`.

[13] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of ACM SOSP*, pages 307–320, Oct. 2007.

[14] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of USENIX OSDI*, pages 49–60, Nov. 1994.

[15] Google. Google Apps: Gmail, Calendar, Docs and more. `http://apps.google.com`.

[16] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. *SIGOPS Operating Systems Review*, 21(5):155–162, 1987.

[17] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group Commit Timers and High Volume Transaction Systems. In *Proceedings of High Performance Transaction Systems*, pages 301–329, 1989.

[18] Y.-C. Ho. On centralized optimal control. *IEEE Transactins on Automatic Control*, 50(4):537–538, 2005.

[19] Hotmail. `http://www.hotmail.com`.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of ACM EuroSys*, Mar. 2007.

[21] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, pages 314–329, Aug. 1988.

[22] R. Jain, D. M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, Digital Equipment Corp., Sept. 1984.

[23] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of USENIX HOTOS*, 2005.

[24] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communications Review*, 33(2):83–91, 2003.

[25] A. Kuzmanovic. The Power of Explicit Congestion Notification. In *Proceedings of ACM SIGCOMM*, pages 61–72, Aug. 2005.

[26] W. Lin, M. Yang, L. Zhang, and L. Zhou. PacificA: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, Microsoft Research, 2008.

[27] Live Mesh. `http://www.mesh.com`.

[28] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of ACM SOSP*, pages 238–251, Oct. 1997.

[29] Microsoft. Office Web Applications. `http://www.microsoft.com/Presspass/Features/2008/oct08/10-28PDCOffice.mspx`.

[30] Microsoft. SQL Data Services. `http://www.microsoft.com/azure/data.mspx`.

[31] M. Moshayedi and P. Wilkison. Enterprise SSDs. *ACM Queue*, 2008.

[32] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187. ACM New York, NY, USA, 2001.

[33] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of USENIX OSDI*, pages 1–14, Nov. 2006.

[34] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of ACM EuroSys*, pages 289–302, Mar. 2007.

[35] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[36] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):146–160, Feb. 1994.

[37] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE Infocom*, pages 1–12, Apr. 2006.

[38] C. Tang, S. Tara, R. Chang, and C. Zhang. Black-Box Performance Control for High-Volume Non-Interactive Systems. In *Proceedings of USENIX ATC*, June 2009.

[39] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of ACM SOSP*, pages 224–237, Oct. 1997.

[40] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of ACM SOSP*, pages 93–109, Dec. 1999.

[41] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1246–1259, 2006.

[42] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of ACM SOSP*, pages 230–243, Oct. 2001.

[43] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of INFOCOM*, Mar. 2004.