Using Indirect Routing to Recover from Network Traffic Scheduling Estimation Error

Conglong Li, Matthew K. Mukerjee, David G. Andersen, Srinivasan Seshan,
Michael Kaminsky[‡], George Porter[†], and Alex C. Snoeren[†]

Carnegie Mellon University [†]University of California, San Diego [‡]Intel Labs

ABSTRACT

Increasingly, proposals for new datacenter networking fabrics employ some form of traffic scheduling—often to avoid congestion, mitigate queuing delays, or avoid timeouts. Fundamentally, practical implementations require estimating upcoming traffic demand. Unfortunately, as our results show, it is difficult to accurately predict demand in typical datacenter applications more than a few milliseconds ahead of time. We explore the impact of errors in demand estimation on traffic scheduling in circuit-switched networks. We show that even relatively small estimation errors such as shifting the arrival time of at most 30% of traffic by a few milliseconds can lead to suboptimal schedules that dramatically reduce network efficiency. Existing systems cope by provisioning extra capacity—either on each circuit, or through the addition of a separate packet-switched fabric. We show through simulation that indirect traffic routing is a powerful technique for recovering from the inefficiencies of suboptimal scheduling under common datacenter workloads, performing as well as networks with 16% extra circuit bandwidth or a packet switch with 6% of the circuit bandwidth.

1. INTRODUCTION

Increasing demand for network bandwidth in the datacenter has renewed interest in circuit-switched network fabrics employing high-capacity optical [9, 12, 26] or wireless [15, 16, 28] links that offer higher effective throughput at lower cost-per-port than traditional electronic packet switches. Initial proposals suffered from large reconfiguration delays, but recent advances have made circuit-switched networks practical [22]. These hardware advances are complemented by software schedulers such as Solstice [21] and Eclipse [25] that take traffic demand (typically gathered through estimation [3, 12]) over the next scheduling interval as input and produce a set of circuit configurations (which ports can communicate) and their respective durations.

One significant challenge for these scheduling algorithms is that the actual offered load may differ significantly from the estimated demand, leading to both unexpected traffic—which may not be admissible by the circuit configurations computed by the algorithm—as well as slack in the configurations from anticipated demand that never materialized. Without an explicit estimation-error recovery mechanism, a circuit-switched network must either serve unexpected traffic over an existing circuit—if an appropriate one exists—likely leading to congestion, or delay serving it until it can be scheduled in the next interval. In either case, the end result is increased delay, either for the unanticipated traffic, the scheduled traffic, or both. Previously proposed solutions include simply increasing circuit link bandwidth (or, equivalently, derating the network capacity) to absorb unexpected demand, or adding a relatively low bandwidth (e.g., an order of magnitude slower) packet-switched alternative, resulting in a hybrid network fabric [9, 12, 20, 26]. Both options can result in a potentially large provisioning cost for the operator.

In this paper, we consider an alternative recovery approach based on indirect routing. Specifically, we use multi-hop paths in the scheduled circuit configurations to serve unexpected traffic between end points that are not directly connected. The indirect traffic uses the slack in each circuit configuration created by anticipated demand that never arrived. Indirect routing is not a new technique; its use in load balancing is widely documented [8, 10, 13, 14, 18, 27], even in the context of datacenter circuit-switch networking [25]. In the past, however, it has served solely as a means to increase circuit utilization. In contrast, we use indirection to address the problem of demand estimation error recovery.

We present a new indirect routing heuristic, called *Albedo*, to recover from demand estimation error. Albedo employs two optimizations to indirect routing to improve estimation error recovery. First, instead of considering all unexpected demand to be equivalent at the byte level, we consider demand on a flow-by-flow basis. We serve older flows first to minimize flow completion time in an attempt to improve application performance. Second, we use two methods of routing traffic over multi-hop paths: buffered indirection and cut-through indirection. Buffered indirection stores traffic at an intermediate host, effectively allowing indirect paths to bridge two different circuit configurations. Cut-through indirection does not store traffic at an intermediate host, and thus can only use paths available within a single circuit configuration. As we show in our evaluation, both optimizations provide benefit.

The contributions of this work are threefold: (i) a characterization of the demand estimation error problem (§3); (ii) the Albedo indirect routing heuristic for estimation error recovery (§4); and (iii) an evaluation of the recovery techniques via simulation (§5). Our results show that:

- Albedo performs better than existing indirect routing algorithms in the context of estimation error recovery.
- Albedo's indirection offers the same estimation error recovery as a network with 16% extra circuit bandwidth or an additional packet switch with 6% extra bandwidth, for common datacenter workloads.
- Albedo provides more consistent recovery than other techniques across a very wide range of estimation error.
- Albedo complements existing recovery techniques.

2. BACKGROUND

We begin by describing the circuit-switching model we consider, as well as the hybrid packet-switched extensions others have proposed to address its limitations. In that context, we survey the scheduling task and examine prior work in scheduling and indirect routing.

978-1-5090-6386-4/17 $31.00 © 2017 IEEE
DOI 10.1109/ANCS.2017.12

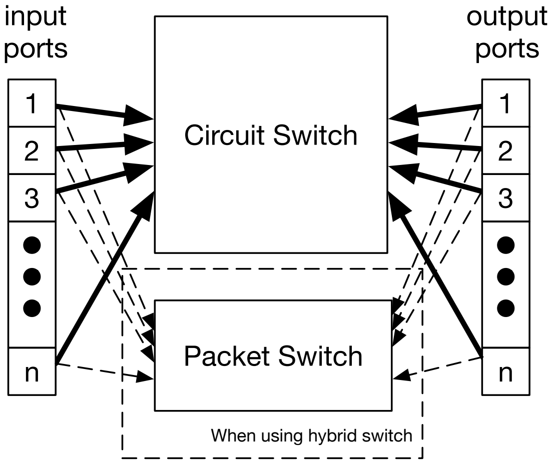


Figure 1: A logical circuit switch model (with optional hybrid packet switching extension).

2.1 Switch model

For the purposes of our study, we consider a single logical circuit switch, although our results can be extended to a physical realization that comprises any number of synchronized, co-scheduled physical circuit switches. Conceptually, a circuit switch connects n input ports to n output ports as shown in Figure 1. Packets at each input port are queued in different virtual output queues (VOQ) [23] based on the destination output port. The circuit switch acts like a crossbar: any input port can connect to any output port, but no input or output port can connect to multiple ports in a given configuration. Moreover, the circuit switch incurs a non-trivial reconfiguration penalty (e.g., 20 μs [20]) when changing input-to-output port configurations.

In order to decrease the impact of the reconfiguration penalty, especially when servicing small or unexpected demand, prior work has considered building an n-port hybrid switch [12, 20, 22, 26] by extending the circuit switch with a packet counterpart, as also shown in the figure. A hybrid switch combines a high-bandwidth (e.g., 100-Gbps or more) low-cost circuit switch with a low-bandwidth (e.g., 10-Gbps) higher-cost packet switch.

In a hybrid model, each input port is simultaneously connected to both the circuit and the packet switch. At any point in time, the packet switch can service multiple VOQs simultaneously at each input port, while the circuit switch can service only the packets queued in a single VOQ (i.e., same destination output port) at each input port. To service the packets waiting in another VOQ, the circuit switch must be reconfigured at the cost of stopping all communication during the reconfiguration. Some technologies allow unchanged circuits to keep servicing packets during the reconfiguration period, but we assume a pessimistic view in which reconfiguration stops the whole circuit switch.

The packet switch is primarily used to deliver small amounts of traffic that might be "difficult" to schedule on the circuit switch efficiently (e.g., 1 KB of data between a source/destination pair that never transmit data otherwise). In the context of this work, we view the packet switch as a form of traffic estimation error recovery mechanism, as it provides flexible any-to-any communication at any time, while the circuit switch is constrained to one-to-one communication based on its current configuration. This flexibility makes it simple for the packet switch to react immediately to unexpected traffic providing straightforward estimation error recovery. However, the cost to provision this additional packet switch fabric is non-trivial; thus, we seek alternate, cost-effective recovery mechanisms that provide benefit for both circuit-only networks as well as hybrid networks.

Input:		
	n	number of switch ports
	D	input demand matrix ($n \times n$)
	r_c	circuit link rate
	δ	circuit reconfiguration time
	W	scheduling window
Output:		
	m	number of configurations
	P_i	circuit switch configuration ($n \times n$)
	t_i	time duration of P_i
Goal:		
		maximize total demand served: $\max \|\| \min(D, \sum_{i=1}^{m} r_c t_i P_i)\|\|_1$ (element-wise min and norm)
Constraints:		
		total time $\leq$ scheduling window: $(\sum_{i=1}^{m} t_i) + m\delta \leq W$

Table 1: Formal description of the scheduling problem.

2.2 The scheduling problem

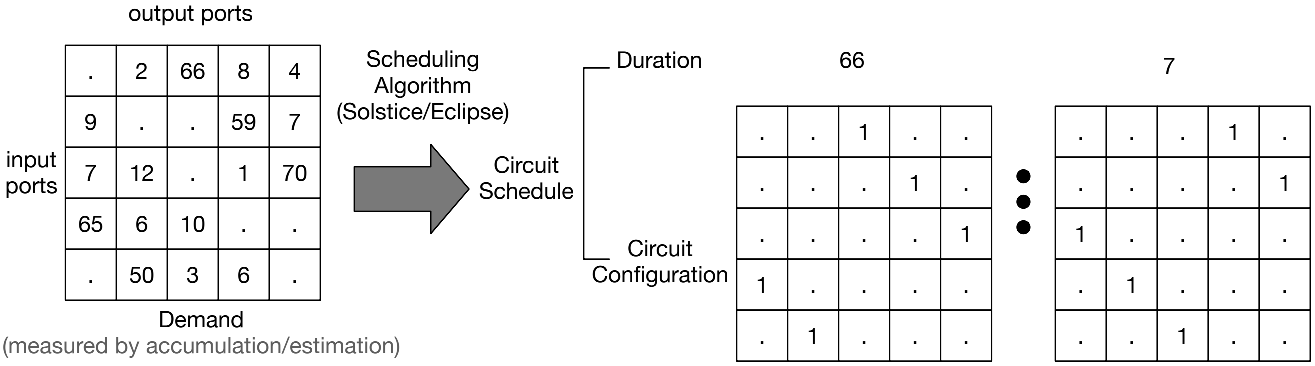
The goal of scheduling traffic on a circuit network is to maximize the amount of traffic we can serve in a set time. Hybrid network scheduling has the same goal, as maximizing the demand served is best achieved by using the much higher bandwidth circuit switch as effectively as possible. Figure 2 illustrates an example of scheduling a 5×5 circuit switch, and Table 1 provides the formalization of the problem. The input of the scheduling algorithm for an n port switch is an $n \times n$ input demand matrix D. The scheduling algorithm produces a set of circuit configurations ($n \times n$ matrices) $\{P_i\}$ and corresponding durations $\{t_i\}$. Each configuration P_i has exactly one 1 in each row and column, referred to as a permutation matrix. The goal of the algorithm is to maximize the amount of demand served, while keeping the total scheduled time (the sum of all configuration durations and the total reconfiguration delay) within a fixed scheduling window W.

Prior work has shown that finding the optimal solution to this problem is NP hard [19], requiring approximation algorithms to solve efficiently. Solstice [21] and Eclipse [25] are two such algorithms that provide nearly optimal schedules using greedy heuristics. Solstice uses a greedy "perfect matching" heuristic based on the Birkhoff-von Neumann (BvN) theorem [6]. Eclipse presents a fast approximate solution to a constructive version of Carathéodory's Theorem for the Birkhoff polytope [7].

Note that the demand matrix D may be provided by either measuring the accumulated demand or by estimating future demand. If D is accumulated, then the schedule produced by the algorithm is for delivering data that has already arrived. If D is estimated, then the schedule produced is for data that is expected to arrive while the schedule is being run. Any error in the estimation means that schedule produced will not match the data that arrived leading to both unexpected demand that can not be served by the schedule, as well as slack (extra capacity) in the schedule that should have been used by demand that never arrived.

2.3 Indirect routing

Indirect routing is the notion of sending data through an additional node before reaching the destination to improve reachability, traffic mixing, or performance. Indirect routing has a long tradi-



.	2	66	8	4
9	.	.	59	7
7	12	.	1	70
65	6	10	.	.
.	50	3	6	.

.	.	1	.	.
.	.	.	1	.
.	.	.	.	1
1	.	.	.	.
.	1	.	.	.

.	.	.	1	.
.	.	.	.	1
1	.	.	.	.
.	1	.	.	.
.	.	1	.	.

Figure 2: Scheduling a 5×5 circuit switch.

tion as a method of load balancing. Valiant load balancing (VLB) is a communication scheme that sends traffic through a randomly selected intermediate node [10]. The benefits of VLB have been shown in packet switch fabrics [8], the Internet backbone [18, 27], and datacenters [13, 14].

The authors of Eclipse also propose the use of an indirect routing heuristic as a performance enhancement called Eclipse++ [25]. Eclipse++ takes the circuit schedule computed by Eclipse and uses indirect routing to serve additional demand, further increasing circuit switch utilization. Eclipse++ iteratively selects the indirect path that offers the largest demand served per spare capacity required. Indirect routing in Eclipse++, however, is solely viewed as an optimization to gain performance over Eclipse, not as a mechanism for recovering from demand estimation error. We show in §4 that there are additional optimizations beyond Eclipse++ that provide significantly better recovery in this context.

3. MOTIVATION

Both Solstice and Eclipse provide fast and efficient heuristic algorithms that deliver high circuit switch utilization. Both systems, however, assume that the scheduler already knows the traffic demand. There are two general approaches to satisfying this assumption: accumulation—record the actual demand generated by the application before transmitting it—and demand estimation—try to predict the demand before it occurs. Unfortunately, each approach has its shortcomings.

Accumulation: Accumulation-based scheduling is simple and accurate. Since the input demands are accumulated from real traffic, the demand to schedule will be 100% accurate. However, this approach requires that a non-trivial accumulation delay must be added to each packet of each flow. This additional latency is often long relative to the duration of latency-sensitive short flows in datacenter traffic [20, 26].

Estimation: Estimation-based scheduling uses hints from applications and/or sophisticated algorithms to predict incoming traffic in order to produce the proper schedule before traffic arrives [3, 12]. While this approach does not introduce additional latency, accurately predicting incoming traffic is difficult. When a predicted flow arrives either earlier or later than anticipated, the computed schedule no longer matches the input, which may add significant latency and degrade performance.

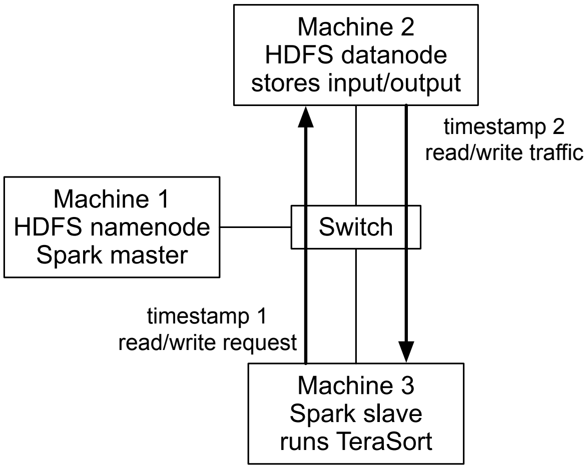


Figure 3: Setup of the traffic estimation experiment for the Spark TeraSort benchmark.

Timestamp difference (ms)	**HDFS read**	**HDFS write**
Average	9.72	10.16
Standard deviation	5.26	8.87

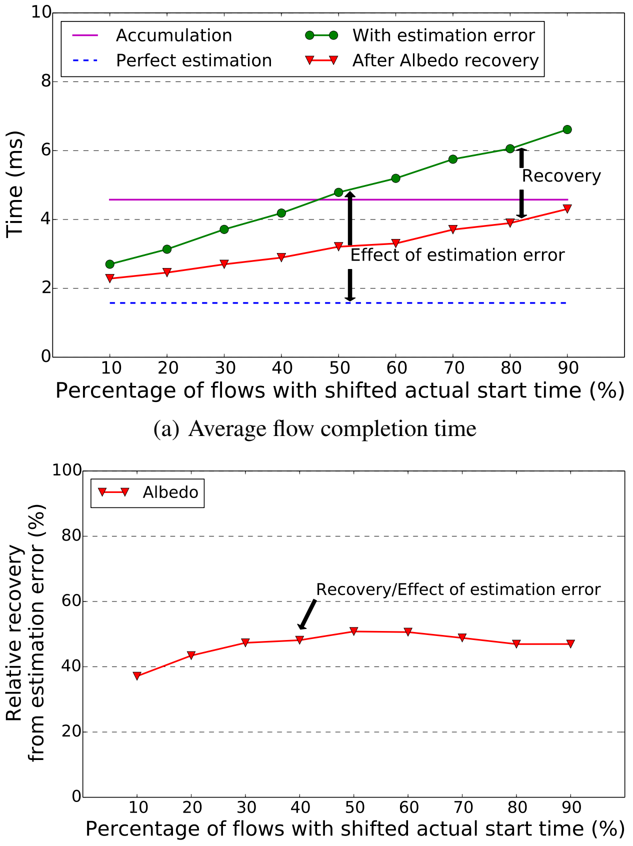
Table 2: Estimation error for the Spark TeraSort benchmark.

Circuit switches are more vulnerable than traditional packet switches to estimation error due to their non-trivial reconfiguration delay (during which data transfer may be disabled). This reconfiguration delay makes it difficult for circuit switches to react on the fly to unannounced traffic caused by prediction error. If the unexpected traffic cannot be served by slack in the existing schedule, the traffic must wait until the next scheduling cycle, incurring latency.

One common source of estimation error is mis-prediction of the flow arrival time, as flows may arrive later than expected due to context switches or cache misses. In the rest of motivation we investigate the possible effect of such estimation error in a real-world application as well as a flow simulator.

3.1 Estimation error in Spark benchmark

To explore estimation error present in a real-world application, we examine the TeraSort benchmark running on Spark [11]. We use the benchmark to sort 20 GB of input, where both the input and output are stored using the Hadoop Distributed File System (HDFS) [24]. Figure 3 shows the experimental setup. Three machines comprise



(a) Average flow completion time

(b) Relative recovery from estimation error

Figure 4: The effect of accumulation and estimation error, and their impact on our routing mechanism Albedo.

the network, connected to the same switch. The first machine acts as the HDFS namenode (master) and Spark master. The second machine acts as the HDFS datanode (slave) where the input data and output data are stored. The third machine acts as the Spark slave which runs the TeraSort benchmark. We instrument HDFS with additional logging to record two timestamps for all HDFS read and write traffic. The first timestamp records when the read or write request is sent by the TeraSort benchmark, and the second timestamp records when the actual HDFS read or write starts on machine 2. We evaluate how accurately we can estimate when a read or write will be transmitted over the network using these timestamps. All three machines are synchronized at microsecond scale using the Precision Time Protocol [2].

We present the average and standard deviation of the difference between the two timestamps for each HDFS read or write access in Table 2. For both HDFS read and write traffic, we observe a high standard deviation relative to the average timestamp difference. This high standard deviation leads us to conclude that using the average timestamp differences would result in significantly error-prone estimates of when data flows will begin.

3.2 Estimation error in flow simulations

Given that the Spark/TeraSort benchmark suggests that accurately predicting the arrival times of flows was difficult, we consider how similar levels of estimation errors would affect circuit switch scheduling. To explore this, we build a flow-based scheduling simulator to generate small and large flows, using the Solstice and Eclipse scheduling algorithms to compute circuit switch schedules. We evaluate schedules using their average flow completion time. We chose this metric because the primary problem arising from estimation error is an increase in the latency of data delivery across the network. For accumulation-based scheduling, we use a 3-ms window to accumulate demand (as in ReacToR [20]). For estimation-based scheduling, we first evaluate the performance if there were no estimation error, then we add estimation error by shifting the start time (by a uniformly random value from 3–6 ms, similar to the amount of estimation error we see in Table 2) for a variable percentage of flows. Other details of the simulation setup are described in §5.1.

Source port	Ports reachable by direct routing	Ports reachable by buffered indirect routing	Ports reachable by cut-through indirect routing
1	2, 4	3	3, 4
2	1, 4	None	3, 4
3	1, 2	4	1, 2
4	3	1	1, 2

Table 3: Ports reachable by direct and 1-hop indirect routing in Figure 5.

Figure 4(a) plots how estimation error affects average flow completion time as a function of the percentage of flows with shifted start time. Compared to perfect estimation (no estimation error), accumulation-based scheduling has much worse performance due to accumulation delay. When many flows have inaccurate start time estimations, however, estimation-based scheduling performs similar or even worse than the accumulation-based scheduling.

To provide reasonable performance using estimation-based scheduling, we need a *recovery technique* that can compensate for traffic estimation error. Traditionally, network designers have provisioned their networks with extra circuit link capacity or propose the use of an additional packet switch for recovery [12,20,22,26]. These additions, however, come at non-trivial cost. This work proposes using *indirect routing* as recovery method. Figure 4(a) also plots how much latency our proposed indirect routing heuristic, Albedo, can recover despite estimation error. With Albedo, estimation-based scheduling performs significantly better regardless of the level of estimation error.

As the latency increase caused by estimation error changes as we vary the amount of estimation error, we propose a metric called *relative recovery from estimation error* to compare recovery techniques that provides fair comparison despite this change. To compute this metric we consider three values: a, the flow completion time of "perfect" estimation-based scheduling without estimation error and recovery technique; b, the flow completion time of estimation-based scheduling with estimation error and without recovery techniques; and c, the flow completion time of estimation-based scheduling with estimation error and recovery techniques. We define the relative recovery from estimation error as $|(b-c)/(b-a)|$. Thus 100% relative recovery means the performance after recovery is as good as the performance without estimation error. Figure 4(b) shows that the relative recovery of Albedo is approximately 50% across a variety of estimation errors.

4. INDIRECT ROUTING

Indirect routing provides a seemingly natural way to combat traffic estimation error. Estimation error always results in one of two scenarios: 1) traffic arrives unannounced, leading to additional latency for this traffic if it cannot be served in the schedule, or 2) expected traffic does not arrive, leading to additional slack (spare capacity) in circuit configurations in the schedule. As estimation error

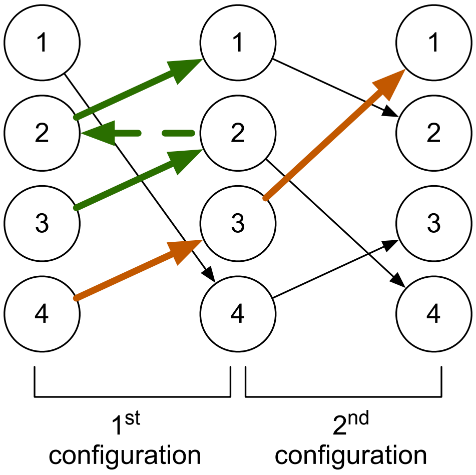


Figure 5: An example of indirect routing in a two-configuration schedule. Green line shows a cut-through indirect routing, and orange line shows a buffered indirect routing.

increases, we expect both of these scenarios to increase. Somewhat counter-intuitively, increasing estimation error leads to increasing slack in the schedule, which leads to increased opportunity to transmit traffic that arrives unexpectedly. Unexpected traffic, however, is most likely not between source/destination pairs configured to communicate within a given schedule, making reachability the key difficulty in serving unexpected traffic. As we will explain through example in Table 3 reachability can be easily extended through indirect routing.

As discussed in §2.3, indirect routing is not a new technique in the context of circuit-switched datacenter networks. As mentioned, Eclipse++ was proposed as an extension of Eclipse in the context of accumulation/perfect estimation scheduling, and thus can be directly applied as an estimation error recovery technique. Our proposed Albedo indirection routing algorithm has two important optimizations beyond Eclipse++.

First, Albedo uses *flow-based* indirect routing. Eclipse++ has no notion of flows, treating each byte to be delivered as equivalent. With Eclipse++, administrators have no way to prioritize the completion of certain flows (e.g., by flow size, timeout value, or more importantly start time). Conversely, Albedo prioritizes serving the flow with earliest start time to reduce average and tail flow completion time. Prioritizing flows with shortest remaining size is known to be the optimal solution for minimizing the average flow completion time. However, the huge size differences between the small and large flows in our experiments make the large flows never get served when using shortest remaining size first policy. Thus we decide to use first-come first-serve policy together with a starvation prevention scheme described in §5.1. When using Albedo in different workload, it is possible that using different scheduling policy could provide better performance, even though the underlying indirection algorithm won't change.

Second, Albedo makes use of two different types of indirect routing: *buffered indirect routing* and *cut-through indirect routing*. Buffered indirect routing uses links across different configurations to form an indirect path. This approach requires buffering at the intermediate hosts on the path. Cut-through indirect routing instead uses links from the same configuration to form an indirect path and thus requires no buffering. Although Eclipse++ only uses buffered indirect routing, we show in §5 further gains in estimation error recovery can be achieved through cut-through indirect routing.

Cut-through indirect routing is not new either [17]. However, we find no prior work applying it in the context of traffic estimation error recovery in circuit-switched datacenter networks. Cut-through indirect routing can be implemented on end hosts or ToR switches. When an end host/ToR receives a packet destined to another host, the network stack can forward it back out. Figure 5 plots examples of buffered and cut-through indirect routing for a 4-port network with 2 configurations, and Table 3 summarizes ports reachability from direct and 1-hop indirect routing (one intermediate host). For each source port, cut-through indirect routing provides additional reachable destinations through indirect paths, as well as the potential for additional paths (with additional capacity) to previously reachable destinations. We conclude that cut-though indirect paths increase both the reachability and opportunity of indirect routing.

Indirect routing has its disadvantages. Clearly, data transferred over indirect paths of length n require n times more bandwidth than data transferred over direct paths (i.e., paths of length 1). However, we propose using indirect routing only in the context of recovery. Once a circuit schedule is selected, we assume that it will be run by the circuit switch. If demand is mis-estimated, the circuit switch may have spare capacity in many configurations that will go unused, making it reasonable to use it somewhat inefficiently with indirect routing. Thus, despite this clear disadvantage in terms of bandwidth consumed, using indirect routing as a recovery mechanism is beneficial.

On the other hand, buffered indirect routing requires buffering packets at intermediate hops. Among all of our experiments, this buffering requires no more than 6 MB per port (2 MB per port on average). It is possible to buffer these packets at the intermediate end hosts. As recent switches provide additional buffer per port at the same magnitude, it is also possible to buffer these packets in the switch [1]. Conversely, Albedo can incorporate a buffering limit in practice to stop buffer indirect routing on a certain path when the buffer is full.

4.1 Albedo indirect routing algorithm

Similar to Eclipse++, Albedo takes the circuit switch schedule computed by the scheduling algorithm and serves additional demand by indirect routing. The difference is that the goal of Albedo is not just to serve more demand, but primarily to mitigate the effect of estimation error. Albedo's indirect routing algorithm is shown in Algorithm 1. Albedo has three inputs: 1) a $n \times n$ non-negative capacity matrix C_j for each circuit configuration (computed by the scheduling algorithm) representing the "spare" capacity of each link in the configuration left over after direct routing (i.e., the amount of data that can be sent over the link during the duration of the circuit minus the amount of data sent from direct routing); 2) the uncompleted flows $\{f_i\}$ and their respective remaining demands $\{d_i\}$; 3) a flag $doCut$ for enabling or disabling cut-through indirect routing. Albedo produces an set of indirect routing decisions $\{I_i\}$ for each flow. Each flow's decisions include pairs of indirect paths and weights $\{(\{P_j\}, w)\}$. The indirect path $\{P_j\}$ is a set of m (0,1)-matrices representing which link in each configuration is used. If $(P_j)_{a,b} = 1$, then the link from port a to port b in the j^{th} configuration is used by the indirect path. The weight w represents the amount of demand served on the indirect path.

Albedo always starts by selecting 1-hop indirect routes for all flows, before trying indirect paths with more hops. Fewer hops are preferable as each additional hop requires more bandwidth as mentioned previously. To minimize the average flow completion

```
Algorithm 1: Albedo indirect routing algorithm
input  : Number of switch ports: $n$,
         $p$ flows $\{f_i\}$ with remaining demands $\{d_i\}$,
         $m$ capacity matrices ($n \times n$) corresponding
         to the circuit configurations: $\{C_j\}$,
         a boolean value to enable/disable cut-through
         indirect routing: $doCut$.
output : Indirect routing decisions for each flow: $\{I_i\}$,
         each flow's decisions include pairs of path and weight:
         $I_i = \{(\{P_j\}, w)\}$
$k \leftarrow 1$
for each $f_i$ do
    $I_i \leftarrow \emptyset$
end
while $k < m$ and $\sum_{j=1}^{m}(\sum_{a=1}^{n}\sum_{b=1}^{n}(C_j)_{a,b}) > 0$ do
    $T \leftarrow$ empty hashtable
    for each $f_i$ with $d_i > 0$ sorted by start time do
        $(src, dest) \leftarrow$ source/destination port of $f_i$
        if $T[(src, dest)] = (\emptyset, 0)$ then
            //no more $k$-hop indirect path for (src, dest)
            continue
        end
        while $d_i > 0$ and $T[(src, dest)] \neq (\emptyset, 0)$ do
            if $T[(src, dest)] =$ NULL then
                //next $k$-hop indirect path for (src, dest)
                if $doCut = True$ then
                    $(\{P_j\}, w) \leftarrow$ earliest $k$-hop
                    buffered/cut-through indirect path from $src$ to
                    $dest$ with nonzero capacity $w$ in each $\{C_j\}$ in
                    the path, or $(\emptyset, 0)$ if no such path exists
                else
                    $(\{P_j\}, w) \leftarrow$ earliest $k$-hop buffered indirect
                    path from $src$ to $dest$ with nonzero capacity $w$
                    in each $\{C_j\}$ in the path, or $(\emptyset, 0)$ if no such
                    path exists
                end
                $T[(src, dest)] \leftarrow (\{P_j\}, w)$
            end
            $(\{P_j\}, w) \leftarrow T[(src, dest)]$
            $w' \leftarrow \min(d_i, w)$
            for each $C_j$ do
                $C_j \leftarrow C_j - w'P_j$
            end
            $I_i \leftarrow I_i \cup \{(\{P_j\}, w')\}$
            if $w > w'$ then
                $T[(src, dest)] \leftarrow (\{P_j\}, w - w')$
            else
                $T[(src, dest)] \leftarrow$ NULL
            end
            $d_i \leftarrow d_i - w'$
        end
    end
    $k \leftarrow k + 1$
end
return $\{I_i\}$
```

time, Albedo uses flow start time to prioritize of each flow. Thus Albedo first selects 1-hop indirect routing for the flow with earliest start time. After selecting all 1-hop indirect routing paths for all flows, Albedo then selects 2-hop indirect routing paths for the flow with earliest start time, and so forth. Each k-hop indirect path consists of $(k+1)$ links, and the source port of each link must be the same as the destination port of previous link in the path. For buffered indirect path, each link comes from a distinct configuration later than all previous links. For cut-through indirect path, using multiple links from the same configuration is allowed as long as the source/destination ports follow the same rule as above. Thus for a $n \times n$ circuit switch schedule with m configurations, buffered indirect routing can have at most $(m-1)$ hops, and cut-through indirect routing can have at most $(mn-1)$ hops. In our Albedo algorithm we only consider cut-through indirect routing with at most $(m-1)$ hops due to the high computational complexity and diminishing returns of the indirect paths with many hops. Albedo stops when all demands are served, when there is no more spare capacity in the configurations, or when all $(m-1)$-hop indirect routing paths are explored.

Technique	Description
+CircuitBW	Extra circuit link capacity
+PacketSwitch	Additional packet switching capacity
Albedo	Proposed indirect routing heuristic (§4.1)
AlbedoNoCut	Albedo without cut-through indirect routing
Eclipse++	Eclipse++ indirect routing heuristic [25]
+CircuitBW+Albedo	First use +CircuitBW, then Albedo
Albedo+PacketSwitch	First use Albedo, then +PacketSwitch

Table 4: Estimation error recovery techniques.

When considering k-hop indirect routing decisions, Albedo uses a hash table to track the last indirect path explored for each (source port, destination port) pair. For each flow, Albedo selects the earliest k-hop indirect path from source to destination port with nonzero capacity w in $\{C_j\}$. The earliest path is determined by the index of the last configuration used in the indirect path. This helps reduce the impact of estimation error on flow completion time. If cut-through indirect routing is enabled, Albedo may use multiple links from a single configuration to form the indirect path. After selecting the indirect path, Albedo determines the corresponding weight by taking the minimum between the flow's remaining demand and the available capacity of the path. Albedo then subtracts this weight from the spare capacity matrices $\{C_j\}$. If the indirect path still has remaining capacity in $\{C_j\}$ after completely serving the demand for the current flow, it will be kept in the hash table with updated capacity to be used by different flows with the same source/destination pair. Otherwise, Albedo searches for the next earliest indirect path with the same constraints in the next iteration.

5. EVALUATION

We evaluate the performance of a variety of recovery techniques summarized in Table 4 to answer the following questions:

1. **Estimation error:** How do different recovery techniques handle different amounts of estimation error? (Albedo recovers from estimation error better than fixed amounts of extra circuit/packet switch bandwidth.)
2. **Extra bandwidth:** For common datacenter workloads, how much extra circuit bandwidth or bandwidth from an additional packet switch is required to perform as well as Albedo? (Albedo performs as well as networks with 16% extra circuit bandwidth or the addition of a packet switch with 6% of the circuit bandwidth.)
3. **Network load:** How does Albedo perform compared to other recovery techniques at different network loads? (Albedo provides much more consistent recovery from estimation despite varying network loads, but has diminishing returns compared to other techniques when less spare link capacity is available at high network load.)

5.1 Simulation setup

5.1.1 Simulation model

We use a 64-port circuit switch with 100-Gbps links and reconfiguration delay of 20 μs across our simulations. Some recovery

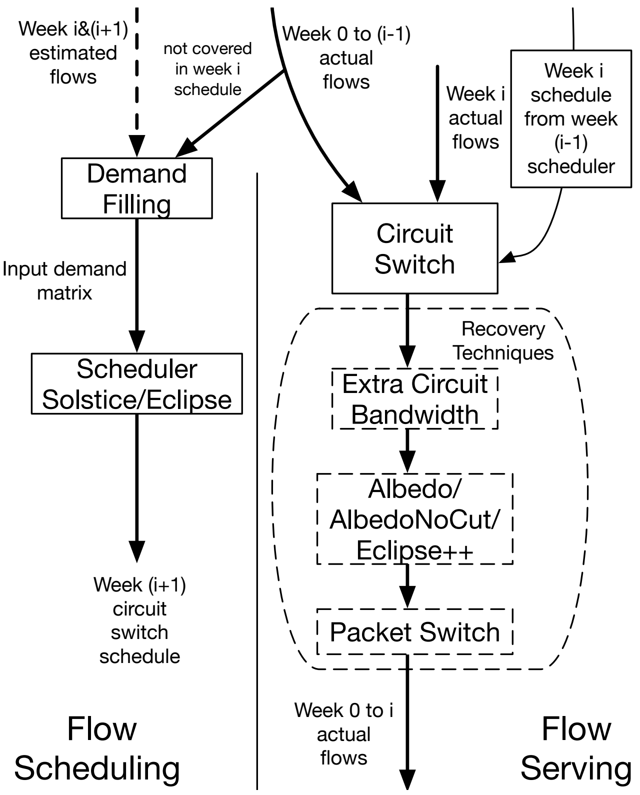

Week i&(i+1) estimated flows
not covered in week i schedule
Week 0 to (i-1) actual flows
Week i actual flows
Week i schedule from week (i-1) scheduler
Demand Filling
Input demand matrix
Circuit Switch
Recovery Techniques
Scheduler Solstice/Eclipse
Extra Circuit Bandwidth
Albedo/ AlbedoNoCut/ Eclipse++
Week (i+1) circuit switch schedule
Packet Switch
Week 0 to i actual flows
Flow Scheduling
Flow Serving

Figure 6: Workflow of week i simulation.

techniques (see Table 4) additionally include a packet switch with by default 5-Gbps links, approximately modeling what is seen in prior work on hybrid datacenter switching [9, 12, 26]. We consider scheduling 3 ms of demand at a time as a "week" (the term used for a scheduling interval in prior work), as in ReacToR [20]. Figure 6 plots the simulation workflow for a single week. During each week there are two tasks running simultaneously: flow scheduling and flow serving. The flow scheduling process produces the circuit switch schedule for next week. At the start of week i, we build the input demand matrix for week $(i+1)$ by combining: 1) the uncompleted flows from weeks 1 to $(i-1)$ that cannot be served by week i schedule; 2) an estimate of flows in week i which will not complete; 3) an estimate of flows arriving in week $(i+1)$. We put these flows into the input demand matrix, annotating them with flow start time for uncompleted flows and an estimated flow start time for estimated flows. To avoid starvation of old flows, we insert flows in order based on their start time, stopping when the input demand matrix is no longer admissible in the 3-ms window. To avoid starvation of small flows, each flow can only consume at most $1/5$ of the weekly bandwidth, unless the input demand matrix is still admissible after filling all the flows. The scheduling algorithm will then compute the schedule for week $(i+1)$ to maximize the amount of input demand served within the 3-ms window.

At the same time, the flow serving process serves the uncompleted flows using the circuit switch and optional recovery techniques. During week i, uncompleted flows from week 1 to i are served using the scheduled circuit switch configurations computed in week $(i-1)$ on a first-come first-served basis based on the flow's start time. To avoid starvation of small flows, each flow consumes at most $1/5$ of the weekly bandwidth, unless there is still spare link capacity after serving all the flows. If there still exist uncompleted flows after filling the circuit switch schedule, a recovery technique from Table 4 may serve them. When using multiple recovery techniques, we first add extra circuit link capacity before using indirect routing, and we use indirect routing before using the packet switch. This is because the circuit switch has strict source/destination pair communication constraints as it must follow circuit switch schedule computed by the scheduling algorithm, whereas the packet switch provides the most flexibility as it can serve any source/destination pair as needed. By default, we use 5% (5 Gbps) extra circuit/packet switch bandwidth for recovery, which we vary in our evaluation of extra bandwidth (§5.3). We use 5% as the baseline since we see similar amount of unexpected demand on average each week.

Throughout our evaluation, we use prior scheduling algorithms Eclipse [25] and Solstice [21]. Note, however, that Eclipse and Solstice vary on their optimization criteria. Eclipse maximizes the demand served within a scheduling window, whereas Solstice minimizes the total scheduled time needed to serve all input demand. To apply the same demand maximization goal to Solstice in our simulations, we stop the algorithm when the total scheduled time exceeds the scheduling window W.

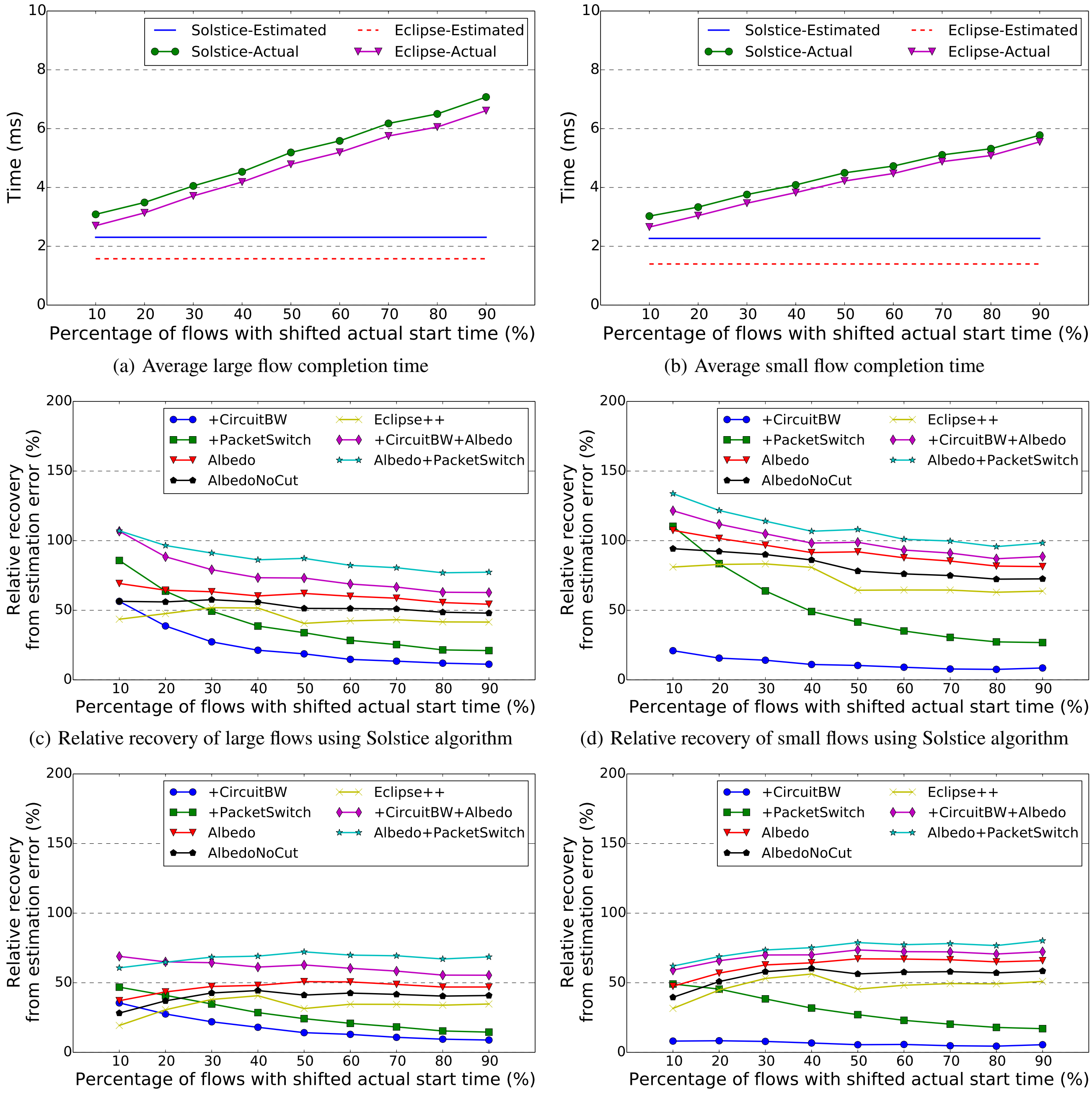
The simulator and the Eclipse/Albedo algorithms are implemented in Go without any optimization. As an online recovery technique, the indirect routing algorithm has to be executed until the start of each scheduling window (when the actual estimation error is confirmed) and has to finish before the end of the window, in order to serve the unexpected demand within the same week. However, it will not become a problem in practice since 1) Albedo's runtime can be improved by better implementation or parallelization, and 2) it is always possible to change the scheduling window size according to Albedo's runtime.

5.1.2 Workload

Each week we generate flows based upon skew and sparsity characteristics from published datacenter traces from the University of Wisconsin [5] and Alizadeh et al. [4]. The Wisconsin study provides two traces of traffic with 500–1000 servers. For each 3-ms window, the traffic matrices from the trace have at most 36 and 85 non-zero elements out of 500 and 1000 servers, respectively. Most hosts only have a few active links, and no host sends large amounts of data to more than 5 other hosts in a window. The Alizadeh work describes the flow behavior and size distributions of a workload combining short query flows and large background flows from production clusters. There are at most 4 concurrent large flows per host in a 50-ms window. The small flows consumes about 10% of the switch capacity, and the large flows consumes about 30%.

Based on the characteristics above, we generate 4 large flows and 12 small flows per source port per week. By default, the total demand of all 16 flows per port consume 30% of the switch capacity, which we later vary in our evaluation of network load (§5.4). We choose 30% network load as baseline based on the Alizadeh work. It is also the largest network load that appears stable for the scheduling algorithms (i.e., with no estimation error the average flow completion time is less than one week). For a given source port, the large flows use 70% of the previously defined 30% network load (split evenly) and the small flows are given the remaining 30% (split evenly). Flow sizes are perturbed by $\pm 10\%$ and each large flow also has a 0.1% probability to become a giant flow with $200\times$ size. At default 30% network load, small flows have average size of 281.25 KB, normal large flows have average size of 1.642 MB, and special large flows have average size of 328.4 MB which makes them unservable within a single week.

The destination of each flow is chosen randomly. Each flow's estimated start time is chosen uniformly at random within the 3-



(a) Average large flow completion time

(b) Average small flow completion time

(c) Relative recovery of large flows using Solstice algorithm

(d) Relative recovery of small flows using Solstice algorithm

(e) Relative recovery of large flows using Eclipse algorithm

(f) Relative recovery of small flows using Eclipse algorithm

Figure 7: Average flow completion time and relative recovery as a function of estimation error for recovery techniques in Table 4.

ms window and is equal to the actual start time when there is no estimation error. With estimation error, we shift the actual start time for 30% of flows by a randomly selected offset between 3 ms and 6 ms. We vary the percentage of flows shifted in our evaluation of estimation error (§5.2). Each flow has a timeout value of $max(24ms, 10 * size/bandwidth)$ so that flows with giant sizes have more flexible timeout values. Any uncompleted flows are dropped after they time out.

We evaluate the network performance over 100 weeks in terms of average flow completion time (excluding flows that time out) and percentage of total demand served (including the partial demand served on flows that time out). We compare different recovery techniques in terms of their relative recovery (as explained in §3.2). To mitigate the effects of simulation start and stop boundaries, we first run 50 "warm-up" weeks where flows are generated at the same rate but not counted in results. After the 50 warm-up and 100 normal weeks, we keep generate new flows at the same rate (not counted in results) until all flows from week 51–150 are completed or dropped.

5.2 Estimation error

As we discussed in §3.2, shifting more flows' start time leads to more estimation error and thus worse flow completion time. Figure 7(a) and 7(b) plot the average flow completion times when we don't shift start times (i.e., perfect estimation; "-Estimated") and when we shift the start time (by [3, 6] ms) for different percentages of flows without using any recovery technique ("-Actual"). When there is no estimation error, both Solstice and Eclipse schedules are able to serve most flows within a 3-ms week. The average completion time of large and small flows are similar as both algorithms try serving large demands first to maximize circuit switch utilization by minimizing the number of reconfigurations (minimizing circuit unavailability due to the reconfiguration delay). As we increase estimation error the average flow completion times also increase. Average large flow completion times increase by 34–320%, and average small flow completion times increase by 33–297%. Even with our baseline estimation error where we only shift 30% of the flows, the average flow completion times increase by at least 66%.

Figure 7(c) and 7(d) plot how different recovery techniques perform in terms of relative recovery as a function of different level of estimation error using the Solstice scheduling algorithm. First, we compare the three different indirect routing heuristics. Albedo always perform better than AlbedoNoCut, which shows that cut-through indirect paths provide additional recovery benefit over just buffered indirect routing. AlbedoNoCut always perform better than Eclipse++, which shows that it is important to prioritize completion of old flows during recovery.

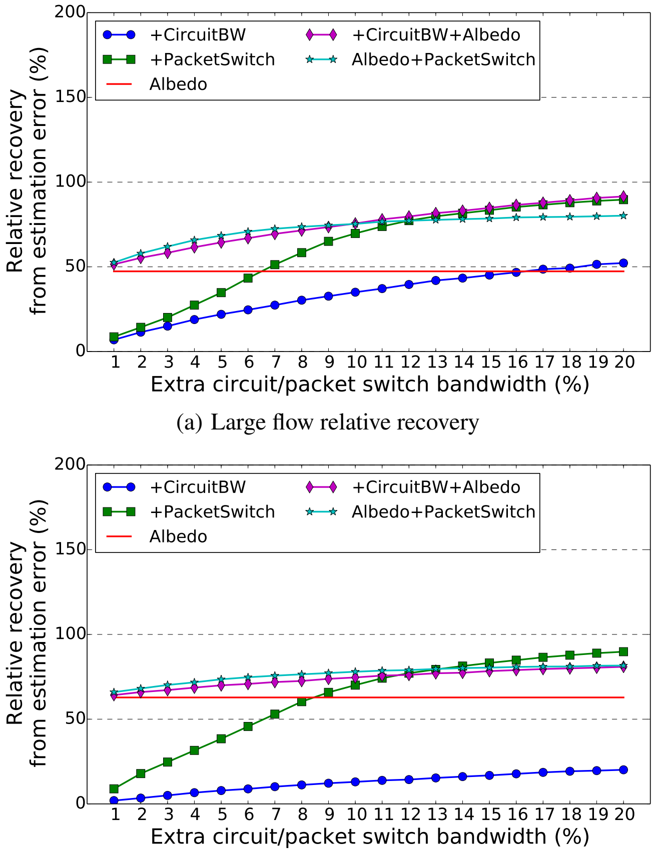
Next, we compare Albedo with extra circuit bandwidth or the addition of a packet switch. Extra circuit bandwidth provides more recovery for large flows than small flows. This is because extra circuit bandwidth is restricted to the sub-optimal schedule produced by the scheduling algorithm, which as mentioned favors serving large demands first. The addition of a packet switch always performs better than extra circuit bandwidth as it doesn't have source/destination pair communication limitations. Both extra circuit bandwidth and an additional packet switch have diminishing returns as the amount of estimation error increases. This limitation is because the extra circuit bandwidth or additional packet switch bandwidth is held constant at 5% of the circuit link bandwidth, regardless of the estimation error. The three indirect routing techniques, by comparison, provide more consistent relative recovery regardless of the amount of estimation error: When estimation error increases spare circuit capacity also increases, allowing for more indirect routing opportunities.

Combining Albedo with extra circuit or packet bandwidth always provides better recovery performance in these experiments. With additional circuit bandwidth, Albedo can recover an additional 5.3–37.5% of the lost flow completion time. With an added packet switch, Albedo can recover an additional 14–37.8% of the lost completion time. With higher estimation error, the benefit of using Albedo in addition to extra circuit bandwidth or an added packet switch is even larger. Effectively, in these experiments indirect routing is complementary to the existing recovery techniques.

Figure 7(e) and 7(f) plot how different recovery techniques perform in terms of relative recovery as a function of estimation error when using the Eclipse scheduling algorithm. The recovery techniques presented provide Eclipse smaller benefit than Solstice. The same takeaways, however, apply to both systems, so we plot only the results for Eclipse in the rest of the evaluation and describe any differences in results between Eclipse and Solstice in text. Since AlbedoNoCut and Eclipse++ perform worse than Albedo in all cases (including the following experiments), for the rest of the evaluation we present Albedo as the only indirect routing recovery technique.

5.3 Bandwidth

We measure how much extra circuit or packet bandwidth is required to provide equivalent recovery when compared to indirect routing for a common datacenter workload with moderate estimation error (30%) by varying the amount of extra circuit or packet bandwidth from 1% to 20%.



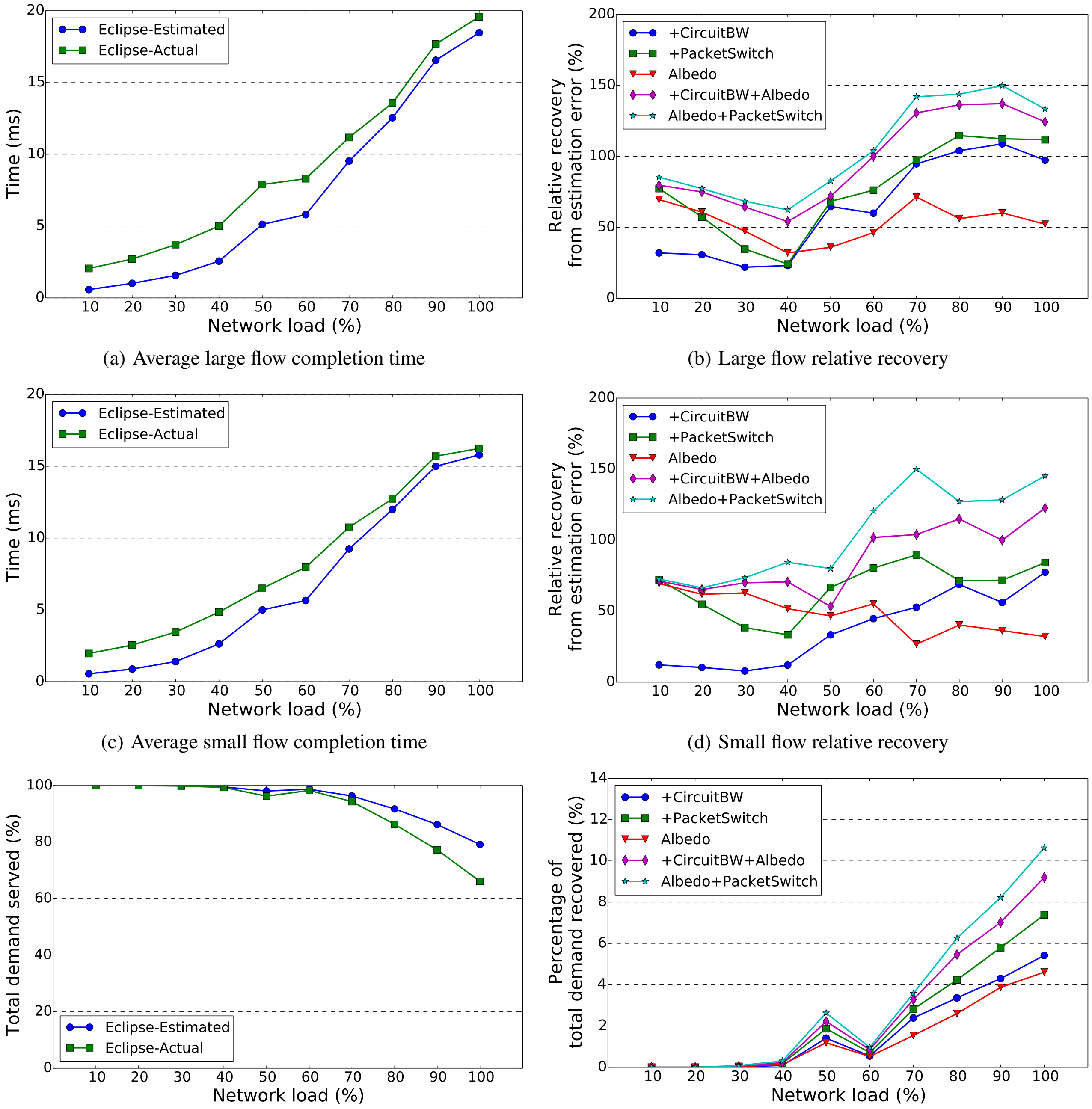
(a) Large flow relative recovery

(b) Small flow relative recovery

Figure 8: Relative recovery as a function of extra bandwidth for the recovery techniques in Table 4 using the Eclipse scheduling algorithm.

Figure 8(a) and 8(b) plot how the different recovery techniques perform with different amounts of extra bandwidth when using the Eclipse scheduling algorithm. For small flows, extra circuit bandwidth never performs as well as Albedo, even with 20% additional capacity. Increasing link capacity in a sub-optimal circuit schedule cannot help unexpected demands reach a large variety of different destinations. To achieve the same small flow recovery performance as Albedo, we need the addition of a packet switch with 8% of the bandwidth provided by the circuit switch. For big flows, Albedo offers comparable performance to 16% extra circuit bandwidth or the addition of a packet switch with 6% of the circuit switch bandwidth. Results for the Solstice scheduling algorithm are similar at 17% and 6% respectively.

Albedo+PacketSwitch actually performs slightly worse than just the packet switch when the packet switch capacity is larger than 12%. This is because Albedo focuses solely on utilizing spare capacity within the circuit switch, ignoring the packet switch. As Albedo always runs before data is sent to the packet switch, this can lead to situations where flows that were simple to deliver on the packet switch take complicated indirect paths on the circuit switch due to Albedo. We leave communication between recovery techniques to future work.



(a) Average large flow completion time

(b) Large flow relative recovery

(c) Average small flow completion time

(d) Small flow relative recovery

(e) Percentage of total demand served

(f) Percentage of total demand recovered

Figure 9: Network performance, relative recovery, and demand served as a function of network load for recovery techniques in Table 4 when using the Eclipse scheduling algorithm.

5.4 Network load

Indirect routing recovery heavily depends on how much spare capacity is available in the circuit switch schedule. Here we evaluate recovery techniques by varying the network load from 10% to 100%. As there are many more flows dropped by timeouts at higher network load due to congestion and scheduling inefficiencies, estimation error affects not only the average flow completion time but also the percentage of total demand served.

Figures 9(a), 9(c), and 9(e) plot how estimation error affects network performance at different network loads when using the Eclipse algorithm. There are two regions of note, low load and high load. At low network load (10% to 40%) the schedule can still serve most of the demand despite the moderate estimation error; only a few flows get dropped by timeouts at 30% and 40% load. With no estimation error, the average flow completion times are roughly 3 ms, similar to a week length. At 10% and 20% load, all flows are completely served. At 30% and 40% load, the percentage of total demand served is at least 99.35%. In this region estimation error mainly affects the average flow completion times. Average large flow completion times increase by 95–251% and average small flow completion times increase by 84–256%.

At high network load (50% to 100%) the network becomes more congested and thus more flows are dropped due to timeouts. The average flow completion times are at least 5 ms (almost double the week length). The percentage of total demand served drops to as low as 79% at 100% load even without estimation error. The backlog of left-over flows begin to take priority over new flows meaning estimation error has much less effect on average flow completion time. Average large flow completion times increase by 6–54%, and average small flow completion times increase by 3–41%. Estimation error does, however, have a large effect on the percentage of demand served. An additional 0.4–13% of the total demand is dropped due to estimation error at high load.

Figures 9(b), 9(d), and 9(f) plot how recovery techniques perform at different network loads using the Eclipse scheduling algorithm. Note that in Figure 9(f) recovery is presented relative to the total amount of input demand rather than relative to the total demand served, as the total demand served is less than 100%. At low network load (10% to 40%), estimation error greatly impacts average flow completion time, but has negligible impact on the total demand served. Albedo always performs better than extra circuit bandwidth in this region. Albedo provides better or similar recovery compared to the addition of a packet switch.

At high network load (50% to 100%), the impact of estimation error on average flow completion time is negligible, yet it's impact on the percentage of total demand served is critical as previously seen. Albedo has less, yet comparable, recovery capability compared to extra circuit bandwidth or the addition of a packet switch. Indirection is less effective because the amount of spare link capacity is greatly reduced when the network becomes congested. Extra circuit bandwidth or an additional packet switch provide better recovery as very high network load leads to demand being queued at nearly all destinations. Any extra bandwidth can be easily consumed by direct routing, which is always a more efficient use of bandwidth than indirect routing.

Adding Albedo to extra circuit bandwidth or an additional packet switch always provide better recovery performance in these experiments. At low network load, extra circuit bandwidth with Albedo can recover at least an additional 30.9% of the average flow completion time. Using Albedo with an additional packet switch can recover an additional 1–51% of the average flow completion time. At high network loads, extra circuit bandwidth with Albedo can recover at most an additional 3.78% of the total demand. Using Albedo with an additional packet switch can recover at most an additional 3.24% of the total demand. This shows that indirect routing is complementary to existing recovery techniques at any network load in these experiments.

6. CONCLUSION

Recent circuit-switch datacenter scheduling techniques can achieve high circuit utilization. Unfortunately, the possibility of real-world demand estimation error breaks many assumptions held by these scheduling techniques, leading to sub-optimal schedules with delayed flow completion times and reduced throughput. Traditional estimation error recovery techniques use costly extra circuit bandwidth or an additional packet switch. We propose an efficient recovery technique, Albedo, based on indirect routing, that requires no additional hardware. Compared to prior indirect routing heuristics, Albedo benefits in this context due to its flow-level view of traffic, as well as its use of cut-through indirect paths. Albedo can recover from estimation error as well as a circuit switch with 16% extra bandwidth or an additional packet switch with 6% extra bandwidth for common datacenter workloads. Albedo also provides additional recovery benefit when combined with these traditional capacity-based recovery mechanisms in most scenarios.

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback. Additionally, the authors would like to thank the National Science Foundation (NSF CNS-1314921 and CNS-1314721), Google (Google Focused Research Award), Microsoft Research, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS) for their support.

7. REFERENCES

[1] Cisco Nexus 9300 Platform Switches Data Sheet. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/datasheet-c78-736967.html.

[2] PTP daemon. https://github.com/ptpd/ptpd/.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. USENIX NSDI*, Apr. 2010.

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

[5] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. ACM IMC*, Nov. 2010.

[6] G. Birkhoff. Tres Observaciones Sobre el Algebra Lineal. *Univ. Nac. Tucumán Rev. Ser. A*, 1946.

[7] C.-S. Chang, W.-J. Chen, and H.-Y. Huang. Birkhoff-von Neumann Input Buffered Crossbar Switches. In *Proc. IEEE INFOCOM*, Mar. 2000.

[8] C.-S. Chang, D.-S. Lee, and Y.-S. Jou. Load balanced birkhoff–von neumann switches, part i: one-stage buffering. *Computer Communications*, Aug. 2002.

[9] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, and X. Wen. OSA: An Optical Switching Architecture for Data Center Networks and Unprecedented Flexibility. In *Proc. USENIX NSDI*, Apr. 2012.

[10] W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[11] Ewan Higgs. TeraSort benchmark for Spark. https://github.com/ehiggs/spark-terasort/.

[12] N. Farrington, G. Porter, S. Radhakrishnan, H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, Aug. 2010.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM*, Aug. 2009.

[14] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, Aug. 2008.

[15] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proc. ACM SIGCOMM*, Aug. 2011.

[16] S. Kandula, J. Padhye, and P. Bahl. Flyways To De-Congest Data Center Networks. In *Proc. ACM HotNets-VIII*, Oct. 2009.

[17] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, Sept. 1979.

[18] M. Kodialam, T. Lakshman, and S. Sengupta. Efficient and robust routing of highly variable traffic. In *Proc. ACM HotNets-III*, Nov. 2004.

[19] X. Li and M. Hamdi. On Scheduling Optical Packet Switches with Reconfiguration Delay. *IEEE JSAC*, Sept. 2003.

[20] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. Circuit Switching Under the Radar with REACToR. In *Proc. USENIX NSDI*, Apr. 2014.

[21] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *Proc. ACM CoNEXT*, Dec. 2015.

[22] G. Porter, R. Strong, N. Farrington, A. Forencich, P.-C. Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *Proc. ACM SIGCOMM*, Aug. 2013.

[23] B. Prabhakar and N. McKeown. On the speedup required for combined input-and output-queued switching. *Automatica*, Dec. 1999.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, May 2010.

[25] S. B. Venkatakrishnan, M. Alizadeh, and P. Viswanath. Costly Circuits, Submodular Schedules and Approximate Carathéodory Theorems. In *Proc. ACM SIGMETRICS*, June 2016.

[26] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time Optics in Data Centers. In *Proc. ACM SIGCOMM*, Aug. 2010.

[27] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone with valiant load-balancing. In *International Workshop on Quality of Service*, June 2005.

[28] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *Proc. ACM SIGCOMM*, Aug. 2012.