

OMNI: OCS Middleware for Network Improvement

Sri Pramodh Rachuri*, Mihir Kestur*, Stefan Schmid†, Anshul Gandhi*

srachuri@cs.stonybrook.edu, mkestur@cs.stonybrook.edu, stefan.schmid@tu-berlin.de, anshul@cs.stonybrook.edu

*PACE Lab, Stony Brook University, USA

†TU Berlin, Germany

Abstract—Datacenter networks face high communication demands when serving data processing workloads like MapReduce. Existing optical fiber connectivity in datacenters alleviates bandwidth bottlenecks to some extent, but they still rely on electrical packet switching (EPS) networks that are limited in bandwidth and suffer from high latencies. Optical Circuit Switching (OCS) offers high-bandwidth, low-latency links compared to traditional EPS networks, but its intermittent connectivity and slow reconfiguration pose significant challenges, especially under dynamic workloads like MapReduce. To remedy this gap, we present OMNI, a demand-aware OCS reconfiguration policy that greedily matches OCS paths to current network demand while being cognizant of the OCS reconfiguration overhead. Extensive experiments on MapReduce using TPC-H workloads show that OMNI consistently outperforms demand-oblivious and existing demand-aware OCS policies, reducing job completion times across diverse workloads, cluster sizes, and dataset scales by as much as 72%, 70%, and 71%, respectively.

I. INTRODUCTION

Datacenter workloads, such as data analytics and machine learning, are often deployed on distributed clusters of servers to handle big data. However, these workloads typically require *significant inter-node communication*, leading to severe network bottlenecks. To handle these large communication demands, datacenter operators have deployed *optical fiber links* over the last decade to interconnect servers and racks, exploiting their superior bandwidth and latency characteristics over traditional electrical links [1]. But, most optical fibers still terminate at Electrical Packet Switches (EPS), which introduce significant latency and bandwidth bottlenecks due to repeated optical-electrical-optical conversions [1], [2].

An emerging solution for addressing such bottlenecks is **Optical Circuit Switching (OCS)**, a technology that enables higher-bandwidth, lower-latency, and contention-free connections between clusters of servers (e.g., racks or pods) when compared to traditional EPS networks [1]. While both OCS and EPS use optical fibers for data transmission, OCS uses a fundamentally different switching mechanism in which direct optical connections/paths are established between source and destination nodes using optical components like mirrors and lenses, bypassing the need for electrical switching and conversions [3]. OCS can reduce data movement time by $10\times$ – $100\times$ compared to traditional EPS [4], [5]. Since OCS’s circuit-switched nature limits the number of simultaneous concurrent connections, in practice, OCS is often used in conjunction with EPS, with OCS accelerating a subset of the links and EPS serving the rest; this is referred to as *Hybrid OCS* [3].

While the link acceleration achieved by OCS is significant, there are challenges in efficiently using OCS in practice:

- **Who to accelerate:** Due to its physical design constraints, OCS can only support a limited number of concurrent optical connections, which restricts the number of links that can be accelerated. As such, *which links to accelerate*, for example, based on the workload demand and network topology, becomes an important question.
- **Reconfiguration delay:** When OCS acceleration for a new link is required, or an existing OCS link needs to be removed, the OCS must reconfigure its optical paths. But, each demand-aware OCS reconfigurations come with overheads and delays, which is usually much slower than the nanosecond-scale switching times of EPS [1], [5], [6].
- **When to accelerate:** Workloads can exhibit dynamic and unpredictable network demands, with intermittent bursts during the execution phases. For example, MapReduce has variable communication patterns, with bursts during shuffle and reduce stages. A Hybrid OCS deployment must thus determine when to accelerate, balancing responsiveness to changing demands with incurring of reconfiguration delays.

Addressing these challenges requires intelligent middleware switching policies that maximize OCS benefits while minimizing associated reconfiguration overheads. Prior work has explored OCS reconfiguration policies and their evaluation, but primarily through simulation or analytical models [7], [8]. Simulations often fail to capture the complexities of real-world deployments, such as the fluctuating, interleaved communication and computation patterns in workload phases, resulting in inaccurate performance predictions. Prior work also focuses on point solutions for specific architectures [4], [9], such as only-OCS setups, only intra-cluster acceleration, or only flat network topology, which is not always the case in practice. Consequently, the gap in current research is largely a practical one; realizing OCS potential in practice requires novel systems engineering and careful framework integration.

In this paper, we address the challenges of integrating OCS with disaggregated workloads and propose a practical solution for maximizing OCS benefits in such frameworks. We present **OMNI, an OCS middleware for network performance improvement**. OMNI greedily reconfigures the optical fabric to match the dynamic network demands of workloads by monitoring network usage and invoking reconfiguration only when necessary. While simple to implement, OMNI is effective and broadly applicable.

To evaluate OMNI, we use MapReduce, commonly used in analytics, machine learning, and scientific computing. Our experiments deploy Apache Hadoop and Apache Spark clusters (of Docker containers) to execute a representative suite of

MapReduce workloads from TPC-H [10], a widely recognized decision support benchmark for big data systems. We emulate OCS behavior using Linux Traffic Control to faithfully model OCS links and reconfigurations. Extensive experiments across different OCS reconfiguration delays, workload profiles, and dataset scales show that our policy consistently reduces MapReduce job completion times compared to existing approaches by 55% on average and by as much as 72%.

II. BACKGROUND

A. Electrical Packet Switching (EPS)

Electrical Packet Switches (EPS) are the traditional connection mechanism in datacenters, receiving data packets from input ports, buffering them temporarily, and forwarding them to appropriate output ports based on routing tables and switching algorithms. While EPS can use physical media such as optical fibers, signals are converted to electrical signals for processing, and routing decisions are made electronically. This optical-electrical-optical conversion can occur multiple times along the path from source to destination in large networks, introducing latency and creating bottlenecks.

B. Optical Circuit Switching (OCS)

Unlike EPS, Optical Circuit Switching (OCS) maintains data in the optical domain throughout the switching process using optical components, giving it several unique properties that differentiate it from traditional EPS networks.

1) *Dedicated Paths*: In OCS, each connection between two servers is established via a dedicated optical path, which remains reserved for the duration of the connection. This contrasts with traditional packet-switched networks, where bandwidth is shared among multiple connections, possibly leading to contention when multiple nodes use the same link.

2) *Directional Paths*: OCS links are inherently directional, meaning that a connection from node A to node B is distinct from a connection from B to A. In the context of OCS configurations, this means that the upload and download connections of a server must be independently accelerated.

3) *One-to-One Mapping*: In this work, we consider OCS switches that support a strict one-to-one mapping between input and output ports. This is a common design choice in many commercial OCS switches, such as those based on 2D MEMS technology that provide dedicated connections from an input port to a single output port at any given time [1], [11]. Conceptually, a switch of this type can be modeled as a bipartite graph between input and output port sets, with the constraint that it must maintain a bijective relationship between these sets. If an OCS top-of-rack (ToR) switch is connected to exactly n nodes, it will have n input ports and n output ports. This allows a maximum of n simultaneous connections and a total of $n!$ unique OCS configurations within the switch.

4) *Reconfiguration Delay*: Most OCS switches are made using micro-electromechanical systems (MEMS) that use mirrors to redirect light beams between input and output ports. When an input port needs to connect to a different output port, the MEMS mirrors must physically move to establish the

new path and disconnect the old one [12]. This reconfiguration process introduces a delay (usually in the order of milliseconds), which we refer to as the “**Reconfig delay**” or “**OCS delay**” [1]. During this reconfiguration, data from the input port cannot be transmitted to any output port through OCS and is instead sent through the EPS fabric [13]. Therefore, every OCS reconfiguration incurs a cost in terms of delay and throughput, which we refer to as the “**Reconfig overhead**”.

In the rest of the paper, we use “connection” to refer to a dedicated OCS path between two nodes; this is not to be confused with TCP/UDP connections used in networking.

C. Hybrid OCS-EPS Fabric

A hybrid OCS-EPS fabric, a common architecture in literature and data center designs [3], combines the benefits of both OCS and EPS. In this architecture, every node is dual-homed, connecting to both an EPS fabric for the basic packet-switched connectivity and a centralized OCS fabric for high-throughput, low-latency data transfers. EPS provides traditional electronic packet switching with paths that are always available but can become congested under traffic. OCS, on the other hand, offers dedicated optical paths that can be reconfigured to meet bandwidth demands between nodes. An important aspect of this hybrid fabric is that OCS links are only available when the ToR switches are reconfigured to establish the necessary paths.

D. Disaggregated Datacenters

Traditional datacenters consist of general-purpose servers with adequate CPU, memory, storage, and networking, but this leads to underutilization due to resource fragmentation [14]. Disaggregated datacenters and public cloud offerings address this by separating resources into distinct pools such as compute, memory, and storage nodes [14], [15]. This architecture allows dynamic resource allocation based on workload requirements. The specialized pools are often located in different racks or clusters within the datacenter, connected over the datacenter’s core network. This separation leads to increased data movement across the network as workloads access resources from different pools, making disaggregated datacenters a good candidate for OCS, as in this work.

III. RELATED WORK

In recent years, there has been significant interest in leveraging OCS to build datacenter networks, as outlined in surveys by Hall *et al.* [16] and Avin *et al.* [3]. Recent research on OCS datacenter networks can be broadly categorized into prototype implementations and simulation or analytical studies.

Among prototype implementations, RotorNet [17] proposes a hybrid OCS architecture but limits OCS connectivity to Top-of-Rack (ToR) switches, making it less relevant to our scenario. Jupiter Evolving [12] demonstrates production-scale OCS with SDN for dynamic reconfiguration but does not address hybrid OCS setups. TopoOpt [9] explores a multi-hop, OCS-only approach with higher-degree connectivity, resulting in an expensive and complex setup. OEHadoop [4] accelerates

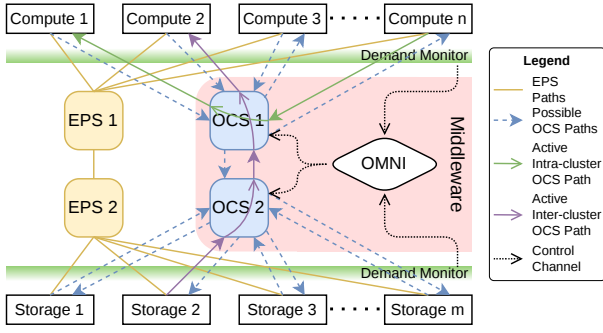


Fig. 1. Illustrating a Hybrid OCS architecture within a disaggregated datacenter showing both OCS and EPS paths. Also shown are two examples of OCS acceleration: (1) intra-cluster (green) and (2) inter-cluster (purple).

MapReduce with OCS but overlooks inter-cluster OCS and reconfiguration overheads, which are critical challenges we focus on in this work. Thus, these works *cannot be easily extended to hybrid OCS networks* in disaggregated setups like ours, despite their valuable insights.

Among simulation and analytical studies, Vermilion [8] introduces a single-hop OCS design with a traffic-aware schedule, improving throughput but not considering hybrid OCS. Chopin [18] and Hanauer *et al.* [19] investigate algorithms for demand-aware datacenter networks, but their focus is on ToR-level OCS and they do not consider individual server connections as in our work. Individual server-level connections are crucial as they capture intra-rack traffic patterns that ToR-level designs completely omit, and these differ significantly from aggregated rack-to-rack connection patterns. Figiel *et al.* [7] is the closest to our work; they present a distributed algorithm, which we refer to as P&R (and evaluate against), for demand-aware hybrid networks. However, the authors only evaluate P&R via simulation, which overlooks practical considerations such as the non-deterministic paths in MapReduce.

Finally, all of the above-cited works assume flat topologies or restrict OCS to only ToR switches, while our setup considers broader OCS integration and evaluates practical feasibility. Flat topologies ignore the hierarchical structure of modern datacenters and fail to capture the complexity of inter-rack OCS communication, whereas ToR-only OCS designs overlook server-level heterogeneity in traffic demands.

IV. SYSTEM DESIGN AND IMPLEMENTATION

We adopt the popular Hybrid OCS architecture with an EPS fallback, suitable for deployment in a disaggregated datacenter. We consider the widely-deployed **MapReduce** as our target workload, which involves significant data shuffling between compute and storage nodes. Figure 1 illustrates a disaggregated datacenter organization, comprising two primary clusters: a compute cluster for CPU/GPU-intensive workloads and a storage cluster for distributed storage. Motivated by the Clos network topology [20] employed by datacenters, we consider that the OCS and EPS switches of each rack are directly connected to the corresponding ToR switches of the other rack, leading to densely interconnected ToR switches.

Figure 1 also illustrates two types of OCS acceleration paths: (1) intra-cluster (green), which are direct optical connections between nodes within the same cluster (compute-to-compute here), and (2) inter-cluster (purple), which are optical connections between compute and storage clusters (storage-to-compute here). Intra-cluster acceleration enables high-throughput transfers for collaborative workloads (like shuffle) or data replication, while inter-cluster acceleration facilitates data movement from storage to compute nodes.

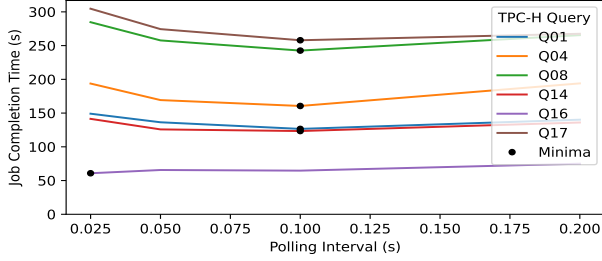
The number of inter-cluster OCS paths is determined by the datacenter’s topology and the available capacity in the aggregation layer above ToR switches [20]. In our work, we consider the aggregation layer providing one inter-cluster OCS path in each direction (between compute and storage clusters), though the design generalizes to arbitrary number of paths.

A. OCS Reconfiguration and OMNI

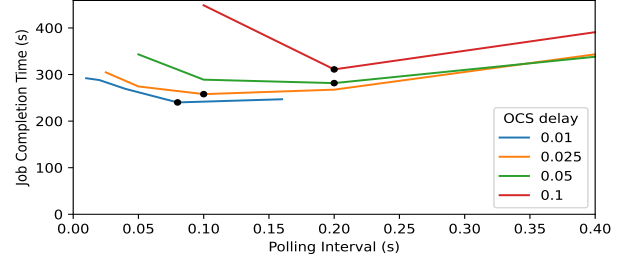
Let vector $\langle a_1, a_2, \dots, a_n \rangle$ denote the configuration of an OCS switch, where n is the number of ports in the switch and a_i is the destination port for the i^{th} port of the switch. If the switch needs to be reconfigured to $\langle b_1, b_2, \dots, b_n \rangle$ in response to, for example, changing workload conditions, then we incur the reconfig cost for only those ports where $a_i \neq b_i$. The problem of finding the optimal OCS configuration has no known polynomial-time solution, scaling as $O(n!)$, where n is the number of ports in the OCS switch [21]. As such, we aim to design a practical and fast heuristic-based approach.

Our system implements our policy, OMNI—OCS Middleware for Network (performance) Improvement. OMNI is a greedy, demand-aware OCS reconfiguration policy that uses demand information to configure OCS paths as needed. OMNI is inspired by greedy maximum weight matching algorithms, but with constraints to handle intra- and inter-cluster conflicts.

OMNI is invoked at regular **polling intervals** at runtime to re-evaluate and potentially reconfigure the OCS paths based on the current demand matrix. It takes in the demand matrix $DM = [d_{ij}]$, where d_{ij} is the amount of data to be transferred from node i to node j , and flattens them into a list of tuples $DL = (i, j, d_{ij})$ representing the source node, destination node, and the amount of data to be transferred. Some elements of DM may be zero, indicating no data transfer between the corresponding source-destination pair; these elements are omitted from DL to save computation. OMNI then sorts this list in descending order based on the amount of data to be transferred d_{ij} and initializes an empty OCS configuration vector C . It iterates through sorted DL , greedily adding source-destination pairs to C , subject to the following conflict constraints. An intra-cluster conflict occurs if the source or destination node of the current pair is already in C . An inter-cluster conflict occurs if the current pair’s direction is the same as any of the inter-cluster OCS paths already in C (since at most one inter-cluster OCS path can exist). If the current pair does not conflict, it is added to C . If it does conflict, OMNI skips that pair and moves on to the next one. When the size of C reaches the maximum number of OCS paths, the algorithm terminates and returns the OCS configuration vector C .



(a) Different TPC-H queries



(b) Different OCS delays

Fig. 2. Job completion time vs. polling interval under OMNI for (a) different queries, and (b) different OCS delays. Dot indicates optimal interval.

Due to the difference in the bandwidth of OCS and EPS paths, already configured OCS paths will use more bandwidth than EPS paths unless they are not actually needed (for example, if there is little data to be transferred). Thus, by design, **OMNI will prefer to keep existing OCS paths if they are actively used, thereby avoiding needless reconfigurations.**

OMNI has time complexity $O(l + m \log l)$, where l is the number of non-zero demand entries and m is the number of iterations to build configuration C . Although l can be up to n^2 , it is typically much smaller due to demand matrix sparsity; in our MapReduce experiments with 15 nodes, only $\sim 5\%$ of entries are non-zero. In practice, m is also small as most reconfigurations only affect a few paths. As a result, **OMNI is effectively linear in l ($\ll n^2$) in real-world scenarios**, making it computationally efficient. In our experiments with 15 nodes, OMNI always completed execution within 0.8ms during cold starts and within 0.3ms for subsequent reconfigurations. OMNI executes **asynchronously** on a separate thread in our implementation, so it does *not* interfere with the main execution. Since MapReduce job completion times are typically on the order of minutes [22], this millisecond overhead is negligible.

B. Optimal Polling Interval under OMNI

The selection of the polling interval is a key factor in balancing reconfiguration overhead and system responsiveness (discussed in Section I). Figure 2(a) shows the MapReduce job completion time as a function of polling interval for different TPC-H queries under OMNI; see Section V for experimental setup. Here, we fix the OCS reconfiguration delay to 25ms. The results suggest that the **optimal polling interval (0.1s) that minimizes job completion time does not change much between queries**. This is likely because the reduction in job completion time due to acceleration is largely dictated by the network bandwidth and availability, with the characteristics of the individual queries having limited impact. We can thus profile a single query and set the optimal polling interval rather than finding the optimal interval for each query separately. We also found that both the slowest polling interval (0.2s) and the fastest polling interval (0.01s) may lead to increased job completion times. This indicates that increasing the frequency of OCS reconfigurations improves the performance of the system up to a point due to faster adaptation, after which the overhead of frequent reconfigurations outweighs the benefits.

Figure 2(b) shows how the optimal polling interval shifts with the OCS delay. **As the OCS delay increases, so does the optimal interval**, suggesting that less frequent reconfiguration is preferable to avoid larger reconfig overhead. In practice, the OCS delay is dictated by hardware. As such, the optimal polling interval does not change for a given hardware.

C. Implementation, Testbed Setup, and OCS Emulation

To deploy a disaggregated datacenter-like environment, we choose Apache Spark as the compute framework and Apache Hadoop Distributed File System (HDFS) as the storage back-end, following the methodology of prior works [4], [14]. All Spark and HDFS components are instantiated as Docker containers on a single physical server, providing an abstraction of multiple nodes, network interface cards (NICs), switches, and network topology. This setup enables fine-grained control over bandwidth and latency between containers, allowing us to accurately model both OCS and EPS paths. Our emulation captures network-level behaviors—circuit mapping, bandwidth allocation, EPS congestion, and reconfiguration downtime—rather than physical-layer optical effects such as signal attenuation or noise accumulation.

1) **Traffic Control and Packet Filtering:** Figure 3 illustrates the flow path of a packet originating from a compute node in our OCS setup. Packets from every node pass through the respective cluster’s switch and the network shaping filters for outgoing traffic. The Demand Monitor monitors the flow to obtain demand estimates. A filter can be represented as a tuple of source, destination, and bandwidth limit. Every cluster’s switch has three types of filters: (i) **Intra-cluster OCS filter** (green): Routes packets over dedicated one-to-one intra-cluster OCS paths between nodes within the same cluster. Each cluster maintains one outgoing filter per node, each with an individually assigned bandwidth. (ii) **Inter-cluster OCS filter** (purple): Routes packets going out of the cluster via inter-cluster OCS paths. Each cluster has at most one such filter, with its own bandwidth allocation. This limits the number of inter-cluster OCS paths to one in each direction, as required. For example, when packets from Storage Node 1 to Compute Node 3 are accelerated via inter-cluster OCS, all other storage-to-compute packets must use the EPS fallback filter. (iii) **EPS fallback filter** (yellow): Acts as the default route, sending all remaining traffic through the EPS (packet-switched) fabric when no OCS filter matches. EPS packets

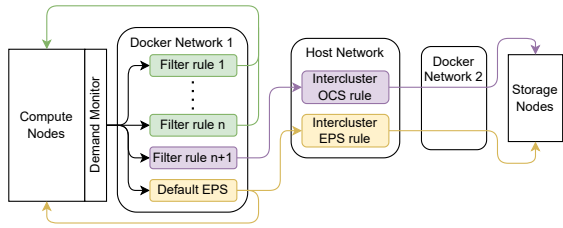


Fig. 3. Illustrating the packet flow in our OCS setup, showing how packets are routed through network shaping filters and how they interact with the Demand Monitor. The figure highlights three types of filters: intra-cluster OCS (green), inter-cluster OCS (purple), and EPS fallback (orange).

going out of the cluster pass through the inter-cluster EPS filter with a separate bandwidth allocation to behave as uplink contention in the datacenter core. All OCS filters are one-to-one between source-destination pairs, while the EPS filter is shared among multiple flows contending for bandwidth.

Figure 3 omits the details of the storage cluster in the interest of space, but it has the same components as the compute cluster. Any packet originating from one cluster but destined for the other cluster must first pass through the host network, which emulates the datacenter core. When packets from inter-cluster OCS filter reach the Host Network (emulating datacenter core), they pass through a separate inter-cluster OCS filter such that they do not cause contention with the inter-cluster EPS filter. But when packets from the default EPS fallback filter reach the Host Network, they pass through a common EPS filter that routes to the destination cluster. This emulates a datacenter-level contention, where all EPS traffic contends for the same uplink bandwidth.

2) *Demand Measurement*: In OCS, accurately measuring demand is challenging as the switches lack visibility into the traffic traversing dedicated optical paths. Unlike EPS, where packet-level monitoring is inherent, OCS requires additional monitoring hardware or estimation techniques. We choose the latter, and consider that each node reports its bandwidth demand to the ToR switch. The Demand Monitor is implemented using custom-written *iptables*’ chains and rules, which track the number of bytes transferred between node pairs.

3) *Middleware and OCS Emulation*: Our centralized middleware collects the demand matrix from both clusters every “polling interval” seconds and is then sent to OMNI (described in Section IV-A) to compute the new OCS configuration using fast numpy matrix operations. The middleware then removes the OCS filters that are no longer in the new configuration and emulates the OCS delay waiting time before applying the new configuration to the OCS fabric. This wait time accurately emulates the OCS reconfiguration delay incurred in real hardware [13]. All filters are implemented using Linux’s *tc filter* command, and the bandwidth limits are enforced using *tc*’s Hierarchical Token Bucket (HTB) queuing discipline. Our methodology does not require any specialized hardware or modifications to network components, making it easy to evaluate different OCS policies in a controlled environment.

Config	#Compute Nodes	#Storage Nodes	#Cores per Container	Mem per Container
Small	5	2	8	20 GB
Default	10	5	4	10 GB
Large	20	10	2	5 GB

TABLE I
CLUSTER CONFIGURATIONS USED IN OUR EXPERIMENTS.

V. EXPERIMENTAL SETUP & METHODOLOGY

Experimental environment. Our evaluation of OMNI is conducted on Amazon Web Services (AWS) using EC2 c5n.metal instances (100 Gbps network, 72 CPU cores, 192 GiB memory). The use of bare-metal instances eliminates virtualization overhead, ensuring full network bandwidth utilization. We use Docker’s *cgroup* feature to pin each container to a set of CPU cores, ensuring exclusive access and preventing interference.

Experimental methodology. We use TPC-H queries as the workload and the DBGen tool to generate different datasets of sizes 5–40GB. Unless otherwise specified, we use a dataset size of 10GB for our experiments. TPC-H is an established decision support benchmark widely used in prior works [10], [14]. The full set of 22 TPC-H queries represents a diverse workload, with some queries being more network-intensive than others due to the volume of the input data they process and the nature of their operations (e.g., joins, aggregations). Furthermore, MapReduce’s bursty and changing communication phases test OMNI’s ability to adapt in a distributed setup.

We consider the *job completion time* as the primary metric of interest. Each experiment is run three times, and the results are averaged to account for variability in job completion times. We pick the best polling interval for each algorithm, unless otherwise specified. In all experiments, we consider the EPS bandwidth to be 100 Mbps and the OCS bandwidth to be 1 Gbps. We set the OCS delay to 25ms unless otherwise specified, and the polling intervals are set to the corresponding optimal values as discussed in Section IV-B.

Table I summarizes the three cluster sizes used in our experiments: Default, Large, and Small. Each configuration varies in the number of compute and storage nodes to evaluate the scalability and robustness of our approach. Unless otherwise specified, we use the Default configuration.

Baseline policies for comparison. To evaluate the performance of OMNI, we implement several popular policies.

- 1) *No acceleration (No Acc)* is a demand-oblivious policy that does not use any OCS paths for data transfers; it only uses EPS paths for all transfers. This policy is instructive in understanding the benefits of OCS acceleration.
- 2) *Naive Round Robin (Naive RR)* is a demand-oblivious policy that cycles through all available OCS configurations in a round-robin fashion similar to RotorNet [17]. It cycles through all available OCS configurations, assigning each one sequentially at every polling interval, without taking into account the amount of data to be transferred.
- 3) *Stage Aware Round Robin (SA RR)* is also a demand-oblivious policy, but it leverages application information for enhanced performance. Specifically for MapReduce

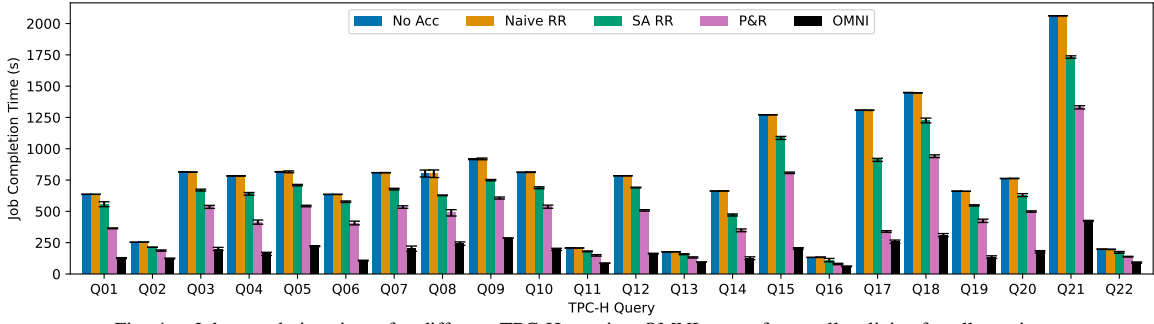


Fig. 4. Job completion times for different TPC-H queries. OMNI outperforms all policies for all queries.

workloads, it finds if the workload is in the map, shuffle, or reduce stage and assigns OCS paths accordingly. In the map stage, most of the data flow occurs from storage to compute nodes, so it only loops through configurations that accelerate from storage to compute nodes. In the shuffle and reduce stages, it only loops through configurations that accelerate from compute to compute nodes.

- 4) *Propose and Reject (P&R)* is a decentralized demand-aware policy proposed by Figiel *et al.* [7] to optimize OCS reconfigurations. It works by having each source node propose an accelerated path to the destination node with the highest demand. The destination nodes will accept the proposal from the source node that has the highest demand, and reject the rest. If it rejects a proposal, it sends a rejection message back to the source node. Then, that source node will propose a path to the next best destination node in its list of destinations. This process continues until all sources have found a destination and no more proposals are made. Unlike OMNI, this policy is unaware of the network hierarchy and assumes a decentralized setup; as a result, it cannot optimize for the best inter-cluster transfer.

We do not have a ground-truth oracle optimal solution to compare against, as workload characteristics and network conditions vary dynamically, making it infeasible to precompute an optimal OCS configuration. Specifically, Spark dynamically allocates MapReduce tasks to executors, which themselves are dynamically assigned to partitions by HDFS; this assignment scheduling is not deterministic. However, by comparing OMNI against a diverse set of baseline policies, we can effectively demonstrate its advantages in various scenarios.

VI. EVALUATION

In this evaluation section, we start with an empirical comparison of job completion times across different policies for all TPC-H queries, followed by an analysis of the achieved network acceleration and reconfiguration overheads. Then, we analyze the impact of OCS delays, cluster configurations, and data scales on the performance of those policies.

A. Comparison with baselines

Figure 4 shows the job completion times for all TPC-H queries under all policies. Standard deviation whiskers are also shown for each bar, though they are barely noticeable due to

low variation between runs. First, we see that Naive RR performs as poorly as the No Acc policy. This shows that *naively employing OCS is not enough to achieve performance gains*. The specific reason here is that the demand-oblivious Naive RR does not have any information about the demand or the workload stage, and so its oblivious usage of OCS is not effective. Next, we see that Stage Aware RR provides lower completion time than Naive RR by taking into account the stage of the workload; averaged across all workloads, SA RR reduces completion time by 16% over Naive RR. While improving over Naive RR by making use of (some) demand information, it still lags behind Demand Aware policies.

Among the Demand Aware policies, OMNI significantly outperforms the P&R policy. This is because P&R is unaware of the network topology and cannot optimize for inter-cluster transfers like OMNI does. Notably, the OMNI policy *consistently outperforms all other policies across all TPC-H queries*, including those with low data transfer requirements such as Q13, Q16, and Q22. This demonstrates the effectiveness of OMNI in optimizing job completion times *regardless* of query characteristics. **Compared to No Acc, Naive RR, SA RR, and P&R, OMNI reduces job completion times, on average, by 72%, 72%, 67%, and 55%, respectively.**

Achieved network acceleration: We now examine how effectively each policy utilizes the accelerated OCS network by analyzing the fraction of data transferred over it. This metric is important because higher utilization of the faster OCS network implies faster data transfers, lower congestion on the EPS network, and possibly improved job completion times. Figure 5 shows that *OMNI transfers the highest percentage of data over the accelerated network* among all policies for all queries. On average, OMNI transfers 89% of the data over the accelerated network across all queries, compared to 27%, 67%, and 77% for Naive RR, SA RR, and P&R, respectively. This ordering is consistent with the performance achieved by various policies in Figure 4, indicating that better usage of the accelerated network leads to improved performance.

We note that while Naive RR does achieve decent data acceleration, it only reduces completion time by at most 1% compared to No Acc (as seen in Figure 4). For example, for Q22, Naive RR accelerates 54% of the data transfer (highest among all queries), but only reduces completion time by less than 0.1% compared to No Acc. This is because of the *reconfiguration overheads* incurred by Naive RR and

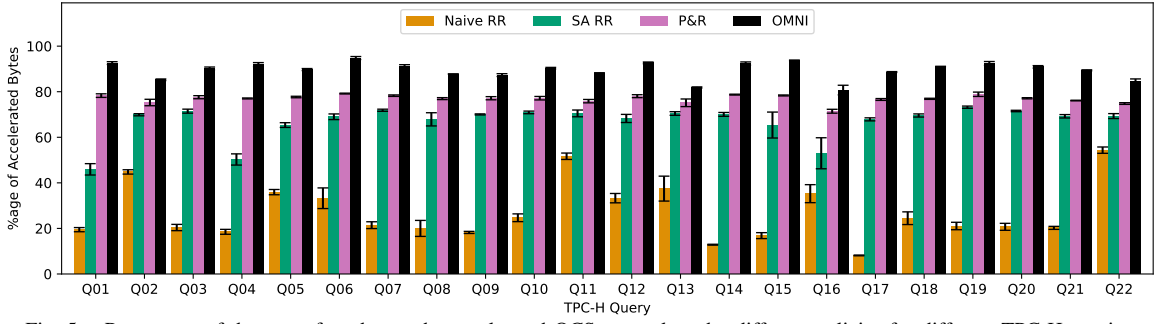


Fig. 5. Percentage of data transferred over the accelerated OCS network under different policies for different TPC-H queries.

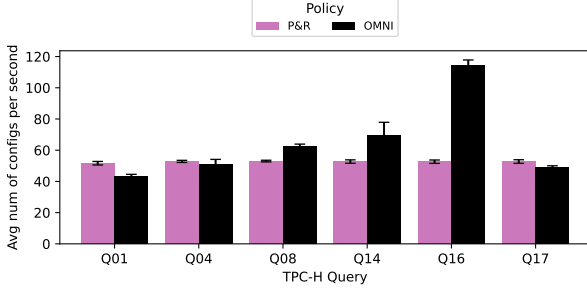


Fig. 6. Average number of reconfigurations per second for different TPC-H queries under P&R and OMNI.

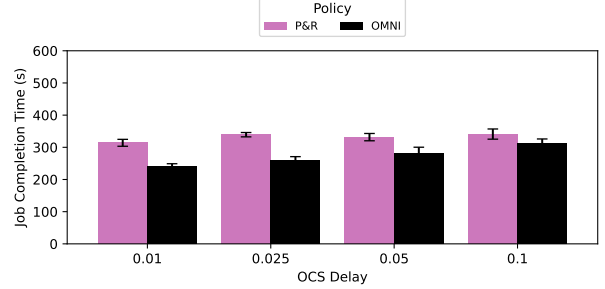


Fig. 7. Job completion times under different OCS delays.

also because Naive RR frequently (through the round-robin policy) accelerates links that are not on the critical path for job completion time. By contrast, OMNI is able to translate the 85% accelerated data for Q22 (from Figure 5) to a 54% reduction in completion time (see Figure 4), demonstrating its effectiveness in utilizing the accelerated network.

Reconfiguration overhead: We now examine the average number of reconfigurations per second for different TPC-H queries, as shown in Figure 6, to understand the reconfiguration overhead incurred by different policies. We omit the No Acc, Naive RR, and SA RR policies to focus on the demand-aware policies, as they are the top performers in terms of job completion times. We also only focus on queries with unique observations in the interest of space.

We do not see any clear trend in Figure 6 between P&R and OMNI in terms of the number of reconfigurations. P&R has a higher number of reconfigs for Q01, Q04, and Q17, while OMNI has a higher number of reconfigs for Q08, Q14, and Q16, with a particularly high number of reconfigs for Q16 ($2\times$ that of P&R). This is because Q16’s minimal data transfer causes rapidly changing MapReduce stages, necessitating frequent reconfigurations. It is interesting to note that OMNI achieves lower job completion times (Figure 4) and a higher percentage of accelerated data transfer (Figure 5) compared to P&R for all queries, despite having *more* reconfigurations for some queries. This indicates that **OMNI makes better reconfiguration decisions (“who to accelerate”), thereby utilizing the OCS network more efficiently**, while trading off the incurred reconfiguration overhead for better performance.

B. Comparison across OCS delays

Next, we analyze how different OCS reconfiguration delays impact performance. Figure 7 shows the job completion times

of Q17 under different OCS delay values (in seconds); we again only focus on demand-aware policies. We pick Q17 as it has moderate data transfer requirements, allowing us to see the impact of OCS delays more clearly. However, we observe similar trends for other TPC-H queries as well. We do not explore OCS delays higher than 100ms as such slow switches are unlikely to be deployed in practice.

We see that OMNI reduces job completion times more effectively than P&R as OCS delay reduces (from right to left), demonstrating that **OMNI can exploit faster OCS reconfigurations better than P&R**. For example, when the OCS delay decreases from 100ms to 10ms, OMNI achieves a reduction in job completion time of 23%, compared to 8% for P&R. Note that both P&R and OMNI do not individually show any significant trend between job completion time and OCS delay; this is because we are using optimized polling intervals (from Figure 2(b)) for the different policies.

C. Comparison across cluster configurations

To understand the impact of cluster configurations, Figure 8 compares the job completion times across cluster sizes: Small, Default, and Large (see details in Table I). We again pick Q17 as a representative query, though we observe similar trends across other TPC-H queries as well. We see similar job completion times for different cluster sizes under the No Acc policy, as the configurations (number of nodes, cores per node, and memory per node) are scaled proportionally, keeping the overall processing power of the compute cluster roughly the same (see Table I). For Naive RR and SA RR policies, the job completion times increase with cluster size, eventually approaching that of No Acc for the Large cluster size. This is because the number of possible configurations increases with the cluster size super-exponentially (as a factorial), reducing

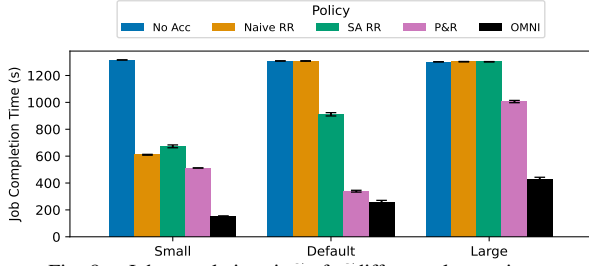


Fig. 8. Job completion times for different cluster sizes.

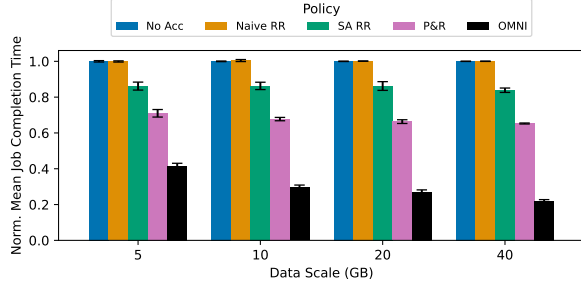


Fig. 9. Mean Job completion times averaged across TPC-H queries at different data scales, normalized to the No Acc policy for each data scale.

the probability of finding an effective configuration. P&R does not show any noticeable trend, but continues to outperform Naive RR and SA RR. **OMNI consistently outperforms all other policies across cluster sizes.** This shows that OMNI scales well with the cluster size and continues to optimize job completion times effectively.

D. Comparison across different data scales

We now analyze the impact of different data scales (size of the dataset used by the queries) on the performance of various policies. Figure 9 shows job completion times across different data scales (5, 10, 20, and 40GB), averaged across TPC-H queries, normalized to the No Acc policy for each data scale. We observe that the ordering of policies remains consistent across data scales, with OMNI outperforming all other policies. The variation in performance between queries (denoted by the confidence interval whiskers) is small. The results show that **OMNI consistently improves job completion times across different data scales** by as much as 71% over P&R, with **increasing benefits for larger data scales.**

VII. CONCLUSION

This paper presented OMNI, a demand-aware OCS reconfiguration policy for disaggregated MapReduce clusters. OMNI addresses key challenges in hybrid OCS-EPS environments, including limited OCS connectivity, slow reconfiguration, dynamic workload demands, and topology constraints. By greedily matching OCS paths to current network demand and tuning reconfiguration frequency, OMNI maximizes OCS benefits in the presence of OCS reconfiguration overheads. Experimental results show that OMNI consistently outperforms existing policies, improving performance across diverse workloads, cluster sizes, and data scales by as much as 72%.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation (NSF) under grants CCF-2324859, CNS-2214980, and CNS-2106434, and by the European Research Council (ERC) under grant 101287293 (FortifyNet), 2026-2027.

REFERENCES

- [1] E. Ding, C. Ouyang, and R. Singh, "Photonic rails in ml datacenters," in *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, ser. HotNets '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 149–159. [Online]. Available: <https://doi.org/10.1145/3772356.3772414>
- [2] C. Kachris and I. Tomkos, "A survey on optical interconnects for data centers," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 4, pp. 1021–1036, 2012.
- [3] C. Avin and S. Schmid, "Revolutionizing datacenter networks via reconfigurable topologies," 2025. [Online]. Available: <https://arxiv.org/abs/2502.16228>
- [4] Y. Tang *et al.*, "Oehadoop: Accelerate hadoop applications by co-designing hadoop with data center network," *IEEE Access*, vol. 6, pp. 25 849–25 860, 2018.
- [5] H. Meyer *et al.*, "Optical packet switching in HPC. An analysis of applications performance," *Future Gen. Computer Systems*, vol. 82, pp. 606–616, 2018.
- [6] A. Juerss and S. Schmid, "Bridge: Optimizing collective communication schedules in reconfigurable networks with reusable subrings," 2026. [Online]. Available: <https://arxiv.org/abs/2605.12766>
- [7] A. Figiel *et al.*, "Distributed construction of demand-aware datacenter networks," in *39th International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.
- [8] V. Addanki *et al.*, "Vermilion: A Traffic-Aware Reconfigurable Optical Interconnect with Formal Throughput Guarantees," <https://arxiv.org/abs/2504.09892>, 2025.
- [9] W. Wang *et al.*, "TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *USENIX NSDI*, Boston, MA, 2023, pp. 739–767.
- [10] "TPC-H Homepage — tpc.org," <https://www.tpc.org/tpch/>.
- [11] Analog Devices, "All-optical switching in transparent networks: Challenges and new implementations," 2016. [Online]. Available: <https://www.analog.com/en/resources/technical-articles/alloptical-switching-in-transparent-networks-challenges-and-new-implementations.html>
- [12] L. Poutievski *et al.*, "Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking," in *Proceedings of the ACM SIGCOMM Conference*, 2022, p. 66–85.
- [13] H. Liu *et al.*, "Circuit switching under the radar with REACToR," in *USENIX NSDI*, Seattle, WA, 2014, pp. 1–15.
- [14] S. P. Rachuri *et al.*, "Optimizing near-data processing for spark," in *42nd IEEE ICDCS*, 2022, pp. 636–646.
- [15] "Amazon ec2 instance types," <https://aws.amazon.com/ec2/instance-type/es/>, [Accessed 02-06-2025].
- [16] M. Nance Hall *et al.*, "A survey of reconfigurable optical networks," *Optical Switching and Networking*, vol. 41, p. 100621, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S157342721000187>
- [17] W. M. Mellette *et al.*, "RotorNet: A Scalable, Low-complexity, Optical Datacenter Network," in *Proceedings of SIGCOMM*, 2017, p. 267–280.
- [18] N. Rozen-Schiff *et al.*, "Chopin: Combining Distributed and Centralized Schedulers for Self-Adjusting Datacenter Networks," in *26th International Conference on Principles of Distributed Systems*, vol. 253, Dagstuhl, 2023, pp. 25:1–25:23.
- [19] K. Hanauer *et al.*, "Fast and heavy disjoint weighted matchings for demand-aware datacenter topologies," in *IEEE INFOCOM*, 2022, p. 1649–1658.
- [20] C. Clos, "A study of non-blocking switching networks," *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.
- [21] H. Liu *et al.*, "Scheduling techniques for hybrid circuit/packet networks," in *Proceedings of ACM CoNEXT*, 2015.
- [22] W. Shi *et al.*, "Smart shuffling in mapreduce: A solution to balance network traffic and workloads," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015, pp. 35–44.