CacheNet: Leveraging the Principle of Locality in Reconfigurable Network Design*

Chen Griner[a], Chen Avin[a], Stefan Schmid[b]

[a] *Ben Gurion University of the Negev, Israel*
[b] *University of Vienna, Austria*

Abstract

Emerging optical communication technologies support the dynamic reconfiguration of datacenter network topologies depending on the traffic they serve. However, to reap the benefits of such demand-aware networks, control logic that quickly learns and adapts to traffic patterns is required. This paper presents CacheNet, a novel approach to efficiently control demand-aware networks. CacheNet consists of two components, a demand-aware *links-cache*, and a demand-oblivious topology. CacheNet leverages temporal and spatial locality in the traffic by managing the reconfigurable links of the optical switches as a *links-cache*. Network traffic, in turn, can be served either by a link from the *links-cache* component or by a demand-oblivious topology component. We study several classic caching algorithms like online LFU and LRU as our caching algorithms, as well as offline optimal caching as a benchmark, and provide an analytical model which captures their performance benefits compared to an all demand-oblivious topology. Our analytical results show that based on the hit ratios and the *links-cache* size, when considering the average packet delay, our hybrid design outperforms a design that is based only on demand-oblivious topology. We also evaluate CacheNet empirically, using both synthetic and real-world traffic traces, confirming the potential of our approach to consider reconfigurable links as a network of *links-cache*.

1. Introduction

Traditional datacenter networks have in common that they rely on a topology which is demand-oblivious, i.e., independent of the current traffic pattern it serves. Recently, reconfigurable optical technologies have introduced an an intriguing alternative to design datacenter networks, allowing to dynamically establish shortcuts, depending on the demand [2, 3, 4, 5, 6, 7]. In particular, such reconfigurable links could be established to support elephant flows or between two racks with significant communication demands. The potential for such demand-aware optimizations is high: empirical studies show that traffic features *locality*,

*A preliminary version of this work was presented in the 2021 IFIP Networking conference [1] .

Email addresses: griner@post.bgu.ac.il (Chen Griner), avin@cse.bgu.ac.il (Chen Avin), stefan_schmid@univie.ac.at (Stefan Schmid)

i.e., traffic matrices are indeed sparse and a small number of elephant flows can constitute a significant fraction of the datacenter traffic [8, 9, 10]. However, designing demand-aware networks is challenging. Existing architectures, including *Helios* [2], *Eclipse* [3, 4] *Solstice* [5], *RE-ACToR* [6] and *Mordia* [7], among others, are based around creating a schedule of reconfigurations for a snapshot of the traffic matrix. More specifically, most existing reconfigurable optical technologies allow to provide dynamic matchings between a set of endpoints (e.g., top-of-rack switches) [11], and throughput can be optimized by cleverly scheduling a sequence of such matchings. This approach however typically requires centralized data collection and introduces a non-trivial computational overhead, which may become a bottleneck in large-scale datacenters.

This paper presents a novel approach to design demand-aware and self-adjusting networks, which is inspired by the success of leveraging the locality principle [12] in other computing systems using caching (e.g., CPU, memory, web caches). That is, rather than aiming to collect information patterns explicitly, we propose an implicit approach in which the different optical switches manage their reconfigurable links as a cache of links. The links in the *links-cache* serve communication requests with very low latency and high capacity, and are adjusted in an online manner, according to the changing demand in the network.

Specifically, we propose *CacheNet*, a hybrid architecture which consists of both demand-aware links, that can be realized as a distributed *links-cache*, and demand-oblivious links. Ideally, the demand-aware component adjusts to changing network demand patterns to serve large flows at lower overhead, while the demand-oblivious component handles any remaining traffic which the *links-cache* cannot handle (e.g., all-to-all shuffle traffic). In particular, the demand-oblivious component of *CacheNet* relies on RotorNet [13], which has been shown to serve shuffle traffic particularly well; however we note that, in principle, any other demand-oblivious network can be used, such as static networks based on expander topologies or electrical networks [14]. We provide a formal analysis of *CacheNet* which allows us to shed light on the optimal partition of demand-oblivious and demand-aware links in the datacenter. We further complement these insights with an empirical evaluation, considering both important synthetic and real-world workloads. Our results reveal that *CacheNet* can greatly benefit from its hybrid design.

To the best of our knowledge, we are the first to establish a connection to explore the opportunities of a distributed *links-cache* to enhance an otherwise demand-oblivious topology. Furthermore, we are not aware on any formal and unifying model which allows us to systematically and analytically quantify to which extent demand-aware cache links are optimally combined with demand-oblivious links.

The rest of the paper is organized as follows, Section 2 introduces our hybrid architecture model, combining demand-aware and demand-oblivious topology components. Section 3 presents a mathematical analysis of our approach, and Section 4 reports on empirical results considering several real-world traces. We discuss limitations and extensions in Section 5, we consider related work in Section 6 and finish with the conclusions in Section 7.

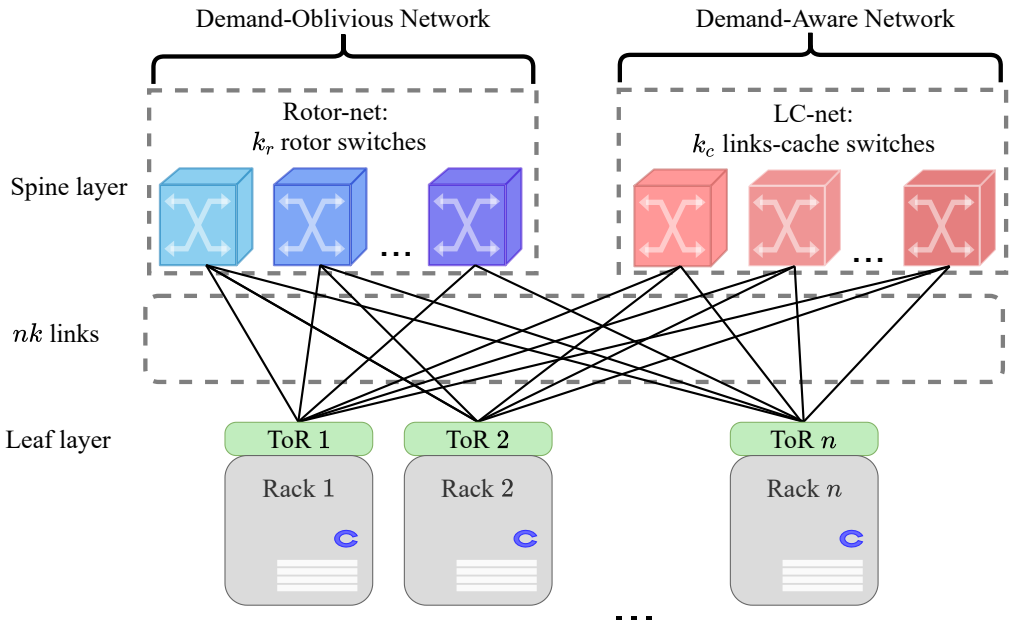


Figure 1: Architecture of *CacheNet* as a two-layer leaf-spine network.

2. CacheNet Hybrid Architecture Model

We consider a hybrid architecture model which will also allow us navigate different configurations and compare trade-offs between demand-oblivious and demand-aware networks as a single system which combines both. In particular, in this paper, we will use RotorNet [13] as the demand-oblivious network; however, rather than using RotorNet directly, as described in this section, we will consider an extended and abstract view of the original RotorNet, henceforth denoted as *rotor-net*. For the demand-aware network, we will use (*links-cache* net) *LC-net*, also described in this section, which is based on our novel distributed *links-cache* approach.

We will simply refer to our hybrid architecture combining *rotor-net* and *LC-net*, as *CacheNet*. To this end, we will assume that for the design of *CacheNet*, we are given a *link budget* (or synonymously edge budget) of m edges (optical links) to serve the communication between n nodes (i.e., possible source or destinations). In a data-center network, sources and destinations could be different ToR switches (as in our empirical traces), but more generally they may represent any type of network nodes (e.g., hosts). Each of these edges is assigned to either the *rotor-net* component or to the *LC-net* component. In the following, we will denote the number of edges assigned to either *rotor-net* or to the *links-cache* as m_r and m_c respectably, and $m_r + m_c = m$. Also note that at all times we maintain exactly m_c edges in the *links-cache*, even before m_c unique edges have appeared in the request sequence. We simply place random edges which are evicted as other edges need to be added by the caching algorithm.

In more concrete terms, we would envision the architecture of *CacheNet* to be that of a two-layer leaf-

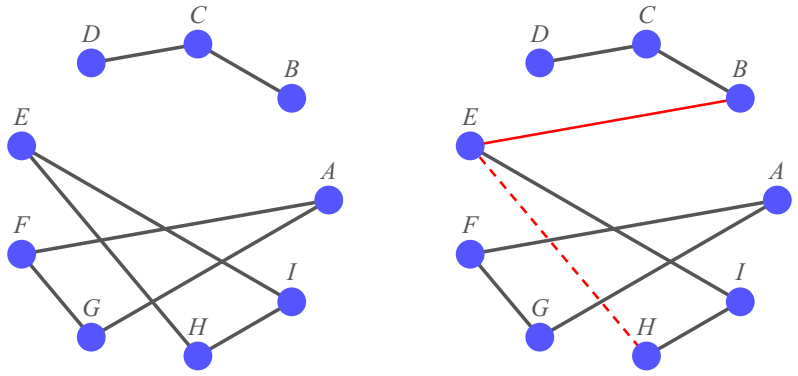
spine network. With n ToRs switches which compose the leaf layer, each with k up-links, which connect to k spine layer switches. Accordingly, we can assume that each switch serves n links and therefore $k = \frac{m}{n}$. Each subsystem, in turn, will have the relative part of the total k switches; In *rotor-net* we have $k_r = \frac{m_r}{n}$ switches and *LC-net* will have $k_c = \frac{m_c}{n}$ switches. Similarly to the edge budget, we can state that our equivalent "switch budget" is $k = k_r + k_c$. A schematic view of our system can be seen in Figure 1. We will now first introduce the two subsystems in turn and then describe *CacheNet*.

2.1. The Demand-Oblivious Network: Rotor-Net

Our demand-oblivious network component builds upon *RotorNet*, proposed in [13]. Since we will make some extensions of this model, as described in the following, we will refer to our version by *rotor-net*.

The original *RotorNet* network is composed of k spine switches and n Top-of-Rack (ToR) switches. Each optical switch in the *RotorNet* network, henceforth called *rotor* switch, has n input and output ports, and independently rotates through a set of input-output configurations, or *matchings*. The set of matchings for each switch is static, and (pre)scheduled in a round-robin manner. The overall collection of these matchings, for all switches, allows *RotorNet* to emulate a complete graph with n nodes, where all $n(n-1)$ possible directed connections are established at some point during a *cycle*. The main cost of this complete graph emulation is the required delay while the network waits for a connection between the two end points to be established. More specifically, at each matching, the *RotorNet* system serves traffic for a certain amount of time, called the *slot* time, δ (i.e., a circuit-hold in which the configuration is not changed); it takes a further amount of time to switch between matchings, denoted as the rotor *reconfiguration time*, R_r. We denote the total time of these two time periods simply as $\tau = \delta + R_r$, the *slice* time. *RotorNet*'s main advantages come form its oblivious nature. Since all the matchings and the round-robin schedule can be calculated in advance, *RotorNet* does not require a sophisticated control plane for topology changes. Furthermore, the *rotor* switch can be designed to have short reconfiguration times, significantly lower than demand-aware switches [15], due to the simple nature of a small set of predefined matchings. Demand-oblivious systems like *RotorNet* are very effective in dealing with uniform traffic; however, when *RotorNet* is presented with more skewed traffic patterns some links may be under-utilized. In this paper we will consider an abstract model of *RotorNet* which we denote by *rotor-net*. In a nutshell, *rotor-net* is simply a reconfigurable network which cycles periodically through a sequence of matchings, in a demand-oblivious manner. *rotor-net* has m_r links and is operated by cycling in a round robin manner through all $n(n-1)$ possible links of the all-to-all *directed* complete graph. In every time slot *rotor-net* connects a set of m_r links and disconnects the previous set of m_r links, until all possible links have been covered in a single full *cycle*[1]. Every such set is a collection of *matchings* and the number of slots that are needed to cover all links is therefore on average given by

[1] Opera [16] proposed a more smooth transition between these sets. For simplicity we ignore this modification.



(a) Cached links (edges) (b) After one reconfiguration

Figure 2: When a request is sent from node E to B using the network in (a) there is a cache miss. *CacheNet* will route the packet using *rotor-net* while also reconfiguring the *LC-net* to the network in (b), remove the edge (E, H) and add (E, B). Future packets are now sent on (b).

$n(n-1)/m_r$. The average cycle time c is given by the slice time τ multiplied by the number of slots (or slices) in a cycle,

$$c = \tau \frac{n(n-1)}{m_r} \tag{1}$$

Since the set of m_r links is built from k_r *rotor* switches, each with matchings of size n, we have that $m_r = nk_r$ and

$$c = \tau \frac{n-1}{k_r} \tag{2}$$

While a link (u, v) is connected, all packets destined from u to v are transmitted, in a single hop with no delay other than the transmission time denoted as t. As in [16] we consider *symmetric* matchings, so whenever a link (u, v) is connected in a *rotor* switch, so is the link (v, u).

2.2. The Demand-Aware Network: LC-net

In general, demand-aware networks are able to capture different demand patterns reconfiguring the network to better suite the demand, hence *"demand-aware"*. Our demand-aware network component is treated as a cache of links and denoted as *links-cache net* or for short *LC-net*. *LC-net* has a total budget of m_c links and each link in the *links-cache* is either connected and ready to be used for packet transmission, or it is being reconfigured, and therefore currently unavailable. In order to capture demand patterns *LC-net* operates in a similar manner to a traditional cache, in the sense that when a link (u, v) is connected (i.e., in the *links-cache*), all packets destined from u to v are transmitted, using a single hop with negligible delay (other than the transmission time t). We keep the *links-cache* symmetric such that when a link (u, v) is in the *links-cache*, so is the link (v, u). When a packet from u to v is sent and the corresponding physical link (u, v) is in the *links-cache*, we have a *cache hit*. In Figure 2 (a) & (b) we can see a simplified example of

Algorithm 1 *CacheNet* packet forwarding

1: Upon a packets from source u to destination v
2: **If** (u, v) exists in the *links-cache*
3: Send packet on the direct link (u, v)
4: **Else** ▷ no such link in the *links-cache*
5: Wait for *rotor-net* to reach a matching with (u, v)
6: Send packet on link in *rotor-net* (u, v)
7: Update the *links-cache* according to a *links-cache* algorithm $\mathcal{A}$

the operation of a *links-cache*. The network has 9 nodes and an edge budget of 8 and is not fully connected. On the network in Figure 2 (a) packets could be sent from node F to node A immediately but not from node E to B. Assuming that such a connection is needed *CacheNet* first forwards packets from E to B on *rotor-net* and then reconfigures *LC-net* according the the caching algorithm. In our example we see that in Figure 2 (b) the connection (E, H) was removed and the link (E, B) was added to *LC-net*, thus allowing immediate communication between the two nodes. Importantly, the decision of when to reconnect a link (insert it to the *links-cache* and remove another link from the *links-cache*) is left to a *cache replacement policy* (a.k.a. caching algorithm). When a link (u, v) is removed from the *links-cache*, and another link (i, j) is added, a reconfiguration time of R_c is incurred. The caching algorithm at the heart of *LC-net* is essentially a prediction algorithm, it predicts which edge is best added to the *links-cache*. While we test three caching algorithms, LFU, LRU and (offline) OPT that are explained later, we note that any prediction algorithm could be used; including algorithm tailored towards communication networks and interestingly those based on machine learning.

2.3. The CacheNet *System*

We assume that *all* packets are sent either on *rotor-net* or on *LC-net*. When a packet arrives to the *CacheNet* system, if an appropriate *cached* link is available, the packet is sent immediately on that link to its destination in a single hop (denoted as a *cache hit*). Otherwise, (denoted as a *cache miss*), the packet is sent using *rotor-net*, and the system's *links-cache* is updated as necessary. During the reconfiguration time both the new and the old links in the *links-cache* are not usable, and all messages for those links are transmitted using the alternate *rotor-net* system. Algorithm 1 presents the operation of *CacheNet*. Observe that whenever a new message arrives to the system, the *links-cache* is (potentially) updated according to a cache algorithm $\mathcal{A}$. In the current work this could be either LFU, LRU or OPT, but any other caching algorithm could be used instead.

For our analysis, we model the traffic as a sequence σ of communication events (e.g., IP packet transmissions) $\sigma = ((s_1, d_1, t_1), (s_2, d_2, t_2), (s_3, d_3, t_3), \ldots)$, where s_i, d_i represent the source and destination nodes, and t_i represents the time at which transmission of packet σ_i occurred.

Given some edge budget m and a traffic pattern σ, our goal is to find a partition of the edges into m_r

and m_c such that the performance of the network is optimized. The optimization result will depend on several parameters: (i) The demand σ, since certain demand patterns will result in a better hit ratio; (ii) the system parameters including τ; and (iii) the reconfiguration times R_r and R_c, as these affect the efficiency of *rotor-net* and the hit ratio. Further discussion on the reconfiguration times can be found in Section 5.

In the following, we will study both the hit ratio of basic caching algorithms and the optimal partition of the total budget m (to m_r and m_c) that will maximize the performance of *CacheNet* analytically. Our system has two extreme modes of operation. i) A complete demand-oblivious network, that is $m_c = 0$ and $m_r = m$. We denote this as a pure *rotor-net* and the system is essentially the original RotorNet. ii) A complete demand-aware network, that is a pure caching system where $m_c = m$ and $m_r = 0$. We denote this as a pure *LC-net*. The more common and non-extreme mode of operation is for both subsystems to have some non zero edge budget, that is both $m_c \neq 0$ and $m_r \neq 0$. This is the case which we found to be ideal in our empirical tests, which check settings from $m_c = 0$ and $m_r = m$ to $m_c = m$ and $m_r = 0$, allowing us to find the optimal configuration for the given edge budget m.

3. Analysis of CacheNet

In this section we continue to outline the theoretical aspects of *CacheNet*. In particular we discuss the main metric used to evaluate *CacheNet* in this paper, the *effectiveness ratio*, which is the ratio of average delay per packet in between *CacheNet* and pure *rotor-net*. We start by analysing the average delay per packet.

3.1. Average Delay Per Packet, comparing CacheNet *to a pure* rotor-net

We evaluate the performance of *CacheNet* on a ToR-to-ToR network, by analyzing at the *average delay* for a packet from the moment it first reached the source ToR, until it arrives at its destination ToR. To derive a concise formula for this delay we assume all packets are of the same size and type, and differ only by their timestamps, source and destination nodes. As mentioned earlier, all packets are transmitted using direct single hop connections. We note that *RotorNet* could in general benefit from 2-hops forwarding, in the form of Valiant routing; however, in this paper we ignore this feature, since we only examine the average delay per packet.

We consider the three main elements of the average delay: the average delay for a packet transmitted on the *rotor-net* subsystem, the average delay for a packet transmitted on the *LC-net* subsystem, and the cache hit ratio of the hybrid system, denoted as t_r, t_c and h respectively.

Let us first analyze t_r. Consider a packet to destination u that arrives to the *rotor-net* from source v. According to our setup, the packet would have to wait for a matching which contains the link $\{u, v\}$ before it is transmitted. If the packet is "lucky" this could happen in the next slot, but in the worst case it could take a whole cycle. If we assume that all packets are equally likely to arrive during any of the matchings,

this would mean that on average a message would wait half a cycle time until it is transmitted. Therefore t_r is given by

$$t_r = \frac{c}{2} + t = \frac{\tau}{2}\frac{n(n-1)}{m_r} + t. \tag{3}$$

The expression for t_c is straightforward. If a packet is sent from a source v to a destination u and the link (v, u) is in the *links-cache*, it would allow it to be transmitted directly, along one hop, to its destination, and therefore $t_c = t$. Clearly $t_r > t_c$, thus we would always prefer to send packets to the *links-cache* whenever possible (i.e., the link is in the *links-cache*).

The last component of the average delay is the hit ratio $0 \leq h \leq 1$, namely, the fraction of times that a packet arrived when the correct link is already in the *links-cache* (and so the packet can be sent using the cached links). This is the ratio between the number of packets sent using the *links-cache*, h_c, and the number of total packets sent, i.e. the length of the trace $|\sigma|$:

$$h = \frac{h_c}{|\sigma|}, \quad h \in [0, 1] \tag{4}$$

We note that the particular value of h gained from the experimental results, is implicitly a function of several variables such as the size of the *links-cache* m_c, the reconfiguration delay of *links-cache* edges R_c and the trace σ itself (a sequence of packets). As a result, it cannot be generally summed into in a simple expression and can take any value in the range $[0, 1]$. But, for special cases we reason about its expected value. Let us begin by defining U_σ to be the set of *unique* communicating requests in σ, that is, the set of requests that are active at least once for the duration of the trace σ. A ratio close to a perfect hit ratio of 1, can be naively achieved for any trace where the size of the set of unique communicating edges $|U_\sigma|$, is smaller than m_c. In this case, we don't need to evict any request for the *links-cache*. That is, if $|E_\sigma| \leq m_c$ then the hit ratio h will be very close to 1 for any trace and for any of our caching algorithms. A value close 0 for h is less likely in a real setting, unless the *links-cache* is very small relatively to $|U_\sigma|$. Theoretically, it might be achieved even for a larger *links-cache* by an adversary which always chooses the next request of the trace to be a link which is not currently in the *links-cache*. In a more realistic setting, even for an i.i.d and uniformly distributed trace we expect the hit ratio to converge towards $\frac{m_c}{|E_\sigma|}$.[2] The hypothesis behind the motivation for this work is that traces with a large degree of structure would have a good hit ratio and that trace do have structure. One recent measure of the amount of structure in a trace is trace complexity [10]: the more complex the trace is the less structure it has. We conjecture that traces and traffic that displays *low* complexity has a *higher* potential to reach better hit ratios.

To continue are analysis, we can deduce the average delay per packet in the *CacheNet* system $AD(h)$ as

[2] In short, since by definition an iid uniform trace has no predictability, the action of any online caching algorithm will not be better than a simple a static *links-cache* with m_c arbitrary links. When each new request is chosen uniformly at random from the set E_σ we get that the probability of a cache hit is $\frac{m_c}{|E_\sigma|}$.

a function of the hit ratio h. This is the weighted average of the delay of each subsystem.

$$AD(h) = ht_c + (1-h)t_r = (1-h)\frac{c}{2} + t. \tag{5}$$

Looking at the result for $AD(h)$, we observe that the average delay per packet is a linear function of h, so its behavior depends entirely on the hit ratio. As expected, the better the hit ratio is, the lower the average delay is.

Finally, we would like to compare our system to a baseline system, which is completely demand oblivious, the pure *rotor-net*, with the same link budget m as the total budget of our system. For this pur *rotor-net* we get an expected delay of

$$t_r^* = \frac{\tau}{2}\frac{n(n-1)}{m} + t \tag{6}$$

To compare both systems (*CacheNet* and pure *rotor-net*) and see what improvement can be gained by using *CacheNet*, we define the *effectiveness ratio* as the ratio of the average packet delay of both systems, assuming that t is negligible:

Definition 1 (Effectiveness ratio).

$$\zeta_m(m_r, h) = \frac{AD(h)}{t_r^*} = (1-h)\frac{m}{m_r} \tag{7}$$

The values of the effectiveness ratio can range from 0 to ∞, and desired values lie in the range 0 to 1. A value less than 1 means that *CacheNet* improves on the baseline system in our case, pure *rotor-net*. We would also note that there is a linear relationship between the effectiveness ratio and the hit ratio given m and m_r: Higher hit ratios lead to a lower effectiveness ratio and better performance of our system for any given switch configuration.

3.2. The Effectiveness Plot

Fig. 3 shows an example of what we call the *effectiveness plot*. The plot's goal is to show the potential benefits of using *CacheNet* and by converting *rotor* links to *cache* links. Each such plot considers a specific trace σ and a link budget m (i.e., the total number of links in the system). The X axis (logarithmic scale) shows the size of the *links-cache*, i.e., the $m_c = k_c n$ links in the *links-cache* (recall that a switch has n links). For example when $k_c = 64$, the *links-cache* size is $m_c = 64 \cdot n$ links. The *link* budget m, for each trace is always equal to 128 switches as was used in the original *RotorNet* paper [13], so $m = 128n$. In Section 5.2 and Table 2 we provide concrete examples for the different traces we use when we discuss *rotor-net* scaling in more details.

The effectiveness plot contains three curves. The first curve is the effectiveness ratio which is denoted by the blue line. The second curve, denoted by the red line, is the hit ratio. The third yellow line, is a "reference line" set to $y = 1$. The Y axis measures both the hit and the effectiveness ratios.

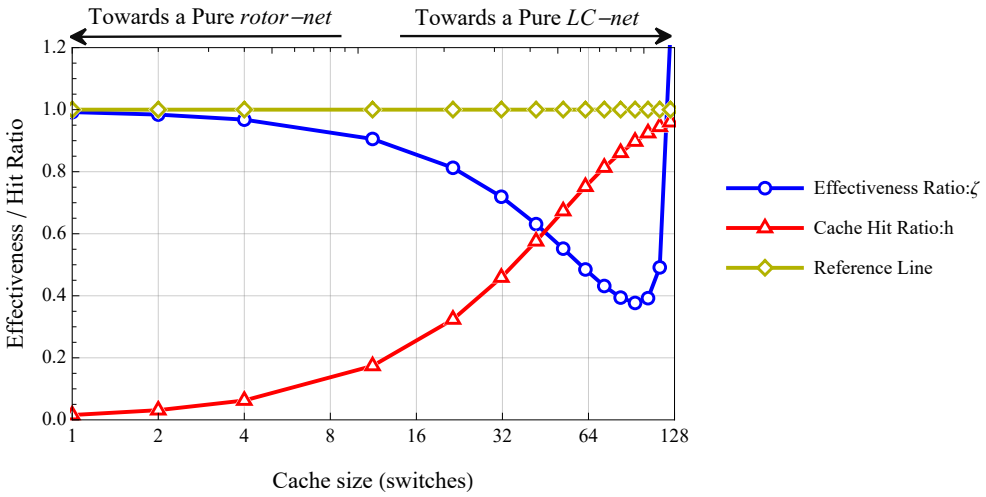


Figure 3: An example of effectiveness plot that shows typical effectiveness and hit ratio curves.

The first observation is that the red line for the hit ratio tends to grow monotonically as the number of switches in *LC-net* increases. This is expected, since the larger the *links-cache* is, more requests can be stored and the likelihood of a cache hit increases. The yellow "reference line" acts as a useful boundary in each plot. If the value of the effectiveness ratio is greater than one, e.g. 1.1 it means that *CacheNet* with the current *links-cache* size, $m_c = nk_c$, has average packet delay that is 10% worse than a *rotor-net*, if the effectiveness ratio is below the line, e.g. 0.9 is means that *CacheNet* outperforms a pure *rotor-net* by 10% in the average packet delay. To produce an effectiveness ratio curve, the values of the empirical hit ratio are fed into the formula presented in Eq. (1). Typically, when k_c is small, close to 1, the effectiveness ratio will be close to 1 as a *CacheNet* with a very small *links-cache* is very similar to a pure *rotor-net*. When k_c is close to 128, which is similar to a pure *LC-net*, the effectiveness will often tend to grow towards ∞, since as we allocate more and more links to the *links-cache* component, any miss will result in a substantial delay for the messages headed towards *rotor-net*, as is apparent from Eq. (3) when m_r approaches 0. An exception to these rules is when the system arrives at a 100% hit ratio when the number of switches is less than 128. Finally, when the effectiveness ratio has one global minimum in the range $k_c \in [1 \dots 128]$, it corresponds to the optimal division of m_r and m_c. This could also happen $k_c = 1$ or even $k_c = 0$ (which is not shown in the figures) when the hit ratio is too low to allow *CacheNet* to outperform a pure *rotor-net* and any number of *links-cache* switches. To summarize, the effectiveness plot shows us when our system outperforms a pure *rotor-net*. When the effectiveness ratio drops below the yellow line our *CacheNet* is better and when it is above it, pure *rotor-net* is better.

4. Empirical Results

We conducted an extensive evaluation of our proposed approach and *CacheNet* in particular. Before presenting our main results, let us first shortly discuss the dataset we used for our evaluation.

4.1. Datasets

In this work we study traces, as detailed in Table 1. Our dataset consists of 17 main trace files from five sources, the first three (ML, Facebook, and HPC) are real world traces, and the others (pFabric, reference points) are synthetic, generated from a simulation: The largest data set was from Facebook datacenter [8]. The set contains three different data center clusters traces of more than 300M entries each, of which we used the first 31M. Each entry contains data about packet size, timestamp, ports, etc, as well as source and destination pods, racks and IPs. We ordered the entries by an increasing timestamp and extract the racks IDs, to create our final traces. The three different clusters, represent three different application types, Hadoop (HAD), a Hadoop cluster, web (WEB), servers that serve web traffic, and database (DB), MySQL servers which store user data and serve SQL quarries. A total of *three* traces where used from this source;

Another interesting data set is the traces of two exascale applications in high performance computing (HPC) clusters [17]: MultiGrid and MOCFE. These represent several different computational kernels of different applications, and show the communication pattern between 1024 CPUs.

Finally, pFabric [18] is a minimalist data center transport design which aims to achieve near optimal flow completion times. For our work we generated packet traces by running the NS2 simulation script obtained from the authors of the paper. We reproduced the scenario: "web search workload" with the pFabric design. We analyzed three packet level traces generated from this simulation, at three different load levels, $0.1, 0.5$ and 0.8. The greater the load, the greater the number of flows in the simulation. In the pFabric simulation flows arrive according to a Possession process. When a flow arrives the source and destination nodes are chosen *uniformly* at random from a set of 144 different IDs. The traces in this paper are commonly used datasets in the literature and are a representative set for a spectrum of traces. They are widely used to describe traffic patterns of HPC [19], Facebook [8, 20], and other workloads [10, 16]. In general, we note that there is a shortage of datasets of this kind, i.e., traffic traces, and therefore we are limited with our choice of datasets.

The traces used in our work are of the form $\sigma = ((s_0, d_0, t_0), (s_1, d_1, t_1), (s_2, d_2, t_3), \ldots)$, where t_i represents the time at which a transmission of packet σ_i occurs and s_i, d_i are the source and destination. In the Facebook trace this time value comes directly from the timetamps which is part of the original trace, where each timestamp represents the *second* when the message was transmitted. In other traces t_i represents the index of the message in the trace, that is $t_i = i$ for all packets. We consider the performance of three basic cache replacement policies: Least recently used (LRU), Least-frequently used (LFU), and OPT (optimal). The OPT policy (a.k.a. Belady's algorithm) can be obtained offline and it always discards the item (in our

Table 1: Traces used in to check hit ratios

Type	Nodes	# of entries
Facebook (Rack) DB [8]	324	31M
Facebook (Rack) WEB [8]	157	31M
Facebook (Rack) Hadoop [8]	365	31M
HPC (MultiGrid) [17]	1024	17.9M
HPC (MOCFE) [17]	1024	2.7M
pFabric (load 0.8) [18]	144	30M

case, a link) that will not be needed for the longest in the future [21]. We note here that the reconfiguration times of the *links-cache* were not taken into account when evaluating the hit ratios. as the timestamp of the traces is not at low enough resolution to allow this. The timestamp units of the Facebook traces are seconds. One second is very large compared to the reconfiguration time of the slowest reconfigurable optical switches, which makes the latter irrelevant. Therefore, to maintain uniformity, we simply assume that the reconfiguration time is zero. Another point is the computational time of the caching algorithm. Since both LRU and LFU are very simple algorithms, we assume that no delay is incurred due to the caching strategy. This might not always be the case, caching strategies based around more complex prediction methods might have a non-negligible impact on the delay of the system. This means that after an event of a cache miss, when a new link is reconfigured (a new link is added to the *links-cache* and an old removed), it will be ready for use by the time a packet with the same source and destination arrives (unless it was removed from the *links-cache* by then). This replacement assumption increases the hit ratios in the results, and therefore, the results should be viewed as an *upper* bound on the performance of *CacheNet*, further discussion of this can be found in subsection 5.3[3].

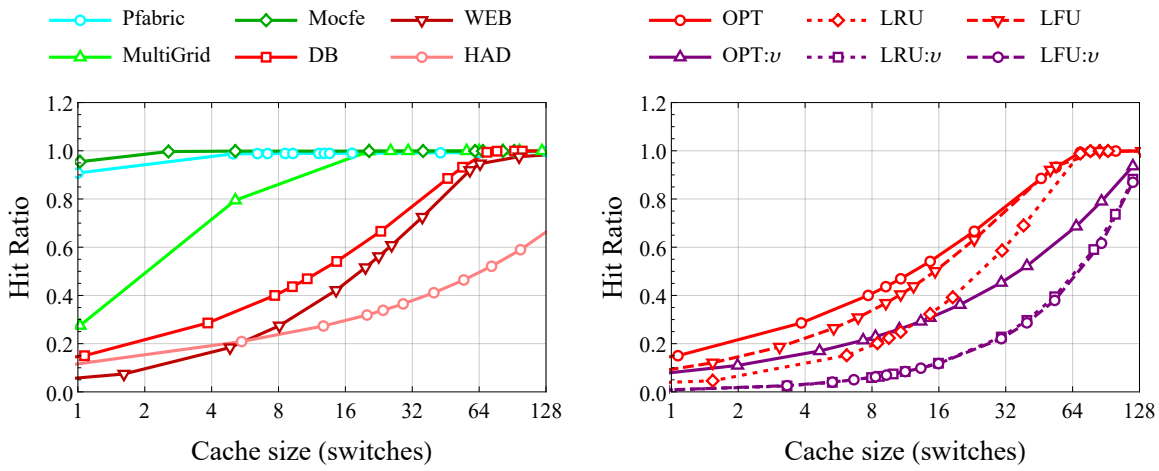
4.2. Simulation Setup

To evaluate the hit ratio for different traces, we simulated a *links-cache* for each of the caching algorithms and every trace described in Section 4.1 using Wolfram Mathematica. In each simulation, the *links-cache* starts with a random set of links. The simulation then runs over the entire trace, checking each request if it is part of the current cached links or not. If it is, the simulation outputs a cache hit; otherwise, it outputs a cache miss and updates the *links-cache* according to the tested cache policy. In turn, when a hit ratio for each trace was found, we calculated the effectiveness ratio using the formulas found in Section2.

4.3. Hit Ratio Results

Let us now look at the hit ratios achieved by our three main cache algorithms. We would like to see which is better for which trace, and which traces achieve a better hit ratio overall. Following our setup, it is

[3]We note that, unlike LFU and LRU, OPT is computationally expensive, and we hence use an approximation algorithm. Instead of finding the link that will appear the latest, from the entire *links-cache*, we sample 10 links, and test among those only. This would mean that our OPT is slightly worse than the actual optimal algorithm.



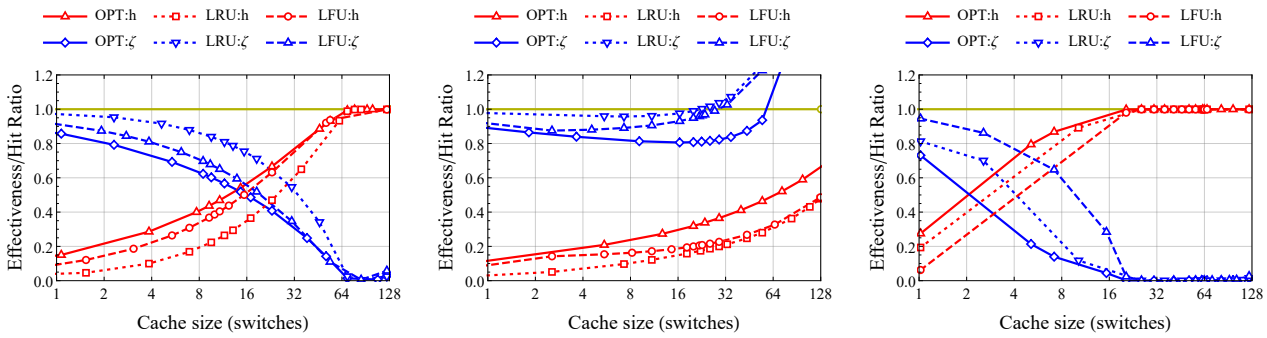
(a) Hit ratios (b) Comparing to uniform

Figure 4: (a) The hit ratios of OPT for six different traces. (b) Hit ratios on the DB trace using the three different caching algorithms LRU, LFU, OPT, as well as adding the hit ratios for the uniform trace $\upsilon(\sigma)$.

clear that a high hit ratio, on a small *links-cache* with few links, will improve the performance of *CacheNet*. Conversely, a low hit ratio may make *CacheNet* inefficient. Figure 4 (a) presents the results of running the OPT caching algorithm. The X and Y axis are represented similarly to the effectiveness plots, the x-axis is the *links-cache* size (m_c) is terms of number of switches of size n and the y-axis is the hit ratio.

Looking at the results we can see that the traces may be divided into two groups of three. The first group is made of the pFabric and two HPC traces, and the other are the three Facebook traces. The first group of traces exhibit very high hit ratios, so high they seem to immediately reach hit ratios of 99% for *links-cache* sizes as low as 5 to 10 switches. This shows that these traces indeed have much structure, that is, a degree of predictability and order. The three Facebook traces exhibit lower hit rates, with the WEB trace showing the highest hit ratio and Hadoop showing the lowest. The Hadoop trace is known to have a very low degree of structure [8] which can explain the low hit ratio. Web servers on the other hand are known to exhibit a high degree of structure [8]. While the OPT policy allows us to investigate some interesting properties, optimal caching requires an algorithm to "know the future" which is generally impossible. Figure 4 (b) presents in addition to OPT the result of LRU and LFU for the DB (database) trace. Both fall short of the performance of OPT, with LFU performing much better in this case. We note that LRU is an algorithm which benefits greatly from bursts of reoccurring elements. The Facebook traces have a 1:30,000 sampling rate [8] so this will likely effect the hit ratio compared to a complete trace. To emphasize how the structure of the trace effects the hit ratio we considered also in the figure the performance of the cache policies on *random* sequences.

Let $\upsilon(\sigma)$ be a uniform i.i.d. sequence trace with the same sources and destination set as σ. Such a trace contains no real patterns or usable structure, and thus is of *maximal entropy* [22]. Any online caching



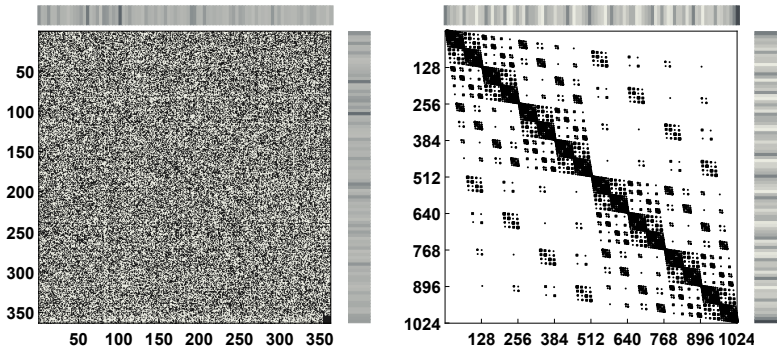
(a) DB trace (b) Hadoop trace (c) MultiGrid trace

Figure 5: The effectiveness plot for three traces with link budget set according to Table 2, $m = 128n$:

algorithm would have poor performance on a uniform trace, since it cannot take advantage of any type of patterns in the trace, such as repetition, or the different frequency of certain elements. The performance of both LRU and LFU should be linear, that is the expected hit ratio is the same as the size of the *links-cache*. Clearly the uniform transformation of the trace causes a significant drop in performance for all algorithms, in particular LFU and LRU show linear performance. OPT is better, this is due to it always being able to predict which of the links is best to remove from the *links-cache*. It is thus able to take advantage even of random patterns that appear in the noise. We present the result for the other traces in Figure 8. We can clearly see that each trace other than the Hadoop trace reach a point of saturation, where the hit ration reaches 100% and this point is also different for each trace. Likely this point reached when the network is overprovisioned and each communicating pair from the trace exists in the caches.

4.4. Performance of CacheNet

We address the performance of *CacheNet* in view of our model. Figure 5 presents the effectiveness plot for three traces along with hit ratio results obtained by using three caching algorithm. Figure 5 (a) of the DB trace shows an almost best case example for *CacheNet*. All tested values for the size of the *links-cache* m_c were able to improve on the performance of *rotor-net*. With all three algorithms reaching nearly 100% improvement with m_c of about 70 switches. These results can be explained as a consequence of a relatively high amount of structure and the low number of rounds at 1.21 meaning a large budget m. Figure 5 (b) shows the results for the Hadoop trace. They present a case were *CacheNet* was able to improve on *rotor-net*, but not nearly as significantly as for the DB trace. In particular we see that LFU and LRU were able to reach about 10% improvement with a small *links-cache* of about 5 to 10 switches, while the improvement brought by OPT is more significant at around 20%. This improvement with a small *links-cache* seems to be a result of the hit ratio growing at a faster rate in the beginning of the curve, where the *links-cache* is small. The growth rate of the hit ratio then slows down. These results with Hadoop are surprising, since the Hadoop trace lacks *significant* structure [8][10], which should lead to negligible improvement. However,



(a) Hadoop (365X365) (B) MultiGrid (1024X1024)

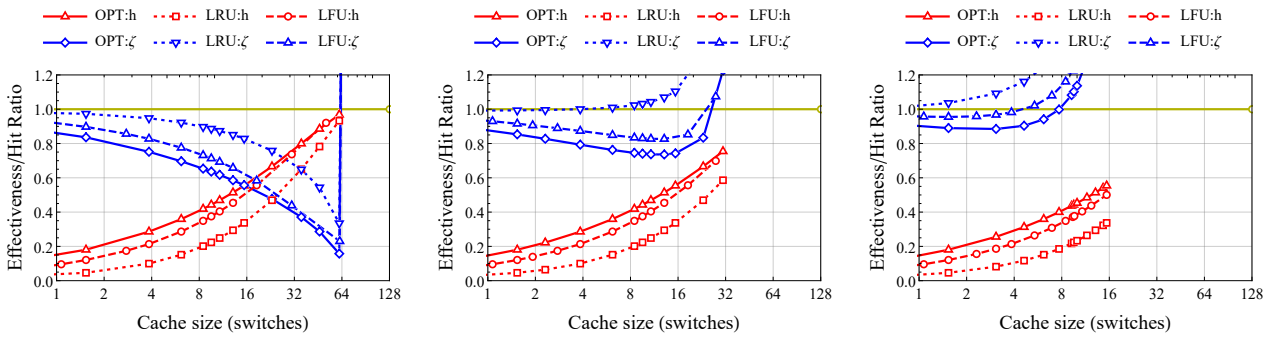
Figure 6: The traffic matrices of Hadoop and MultiGrid

looking at the hit ratio curves where $x < 10$ the hit ratio seems to grow at a faster rate, which may indicate some structure that LFU and OPT are able to take advantage of. This structure could be, for example, the results of a small subset of links which appear at a higher frequency than the rest of the links, that are uniformly distributed. LFU would naturally store those (few) frequent links and gain a better hit ratio. When the size of LFU is increased over the size of this subset the gains in the hit ratio become linear, and as a result, cannot improve the effectiveness ratio. This is marginally supported by the Hadoop traffic matrix in Fig. 6 where there seems to be a set of more active links at the lower right corner of the matrix.

Figure 5 (c) presents an HPC trace of the MultiGrid application [17]. Interestingly the effectiveness plot shows that the hit ratio of LRU is superior to LFU. However, all three algorithms reach a hit rate of about 100% with 20 switches. One possible explanation for the under-performance of LFU is that while the HPC trace distributions are skewed, they are only skewed in the sense that they are sparse; that is, only a small part of the possible communicating pairs appear in the trace. The pairs that do appear in the trace are (relatively) uniformly distributed. Additionally, the number of these pairs is small enough such that a *links-cache* of about 20 switches is able to fully contain it. This is supported by the traffic matrix of MultiGrid shown in 6 (b). The success of LRU indicates burstiness of the sequence which in turn support the need for *cache* switches. Figure 9 presents the effectiveness plot for three more traces.

4.5. The Effect of a Smaller Edge Budget

We so far considered a fixed-size edge budget m, which was based on the scaling the formula discussed earlier where $m = 128n$. It may be interesting to see the behavior of *CacheNet* under other, less favourable, configurations of an edge budget. Figure 7 shows the results of running our simulations with three *relatively* smaller, edge budgets on the DB trace. These edge budgets correspond to a half, a quarter, and an eighth of the number of edges in the original budget of 128 switches. We can see that the results for the DB trace as seen in Figure 5 (a) are promising. However, these represent an edge budget of 128 switches which require 1.21 rounds to complete a cycle. Removing resources from the system, and hence reducing the maximum



(a) $m = 64$ switches (b) $m = 32$ switches (c) $m = 16$ switches

Figure 7: The DB trace is considered to study the effect of different edge budgets on the effectiveness ratio, with three different sizes of the *links-cache*.

LC-net and *rotor-net*, will naturally make *CacheNet* less effective.

Figure 7 (a) for $m = 64n$, i.e., 64 switches, shows that, while the results have not been as good as in the original configuration, *CacheNet* still reaches an improvement of between $80\% - 65\%$, with LFU being worse. The optimal switch allocation is fully *links-cache*. In (b), when $m = 32n$ all algorithms are worse, however LRU now provides no significant improvement at all, while LRU and OPT provide up to 20% improvement of the base *rotor-net*. The optimal switch allocation is now roughly 16 *links-cache* switches, that is, half the system, for each of the caching algorithm. Lastly, in (c) where $m = 16n$ LFU offer only a minor advantage over the pure *rotor-net*, with only OPT providing up to a 10% improvement. The optimal switch allocation is now roughly 8 *links-cache* switches, again, about half the system. The results show how removing resources from cache-net impacts the results. Smaller systems will have a worse performance gain. However, this shows that even with significantly smaller m there is still some gain over the basic *rotor-net*. Finally we note that these results do not mean that a cache-net with a small *links-cache* would not work. A traffic pattern that enables a high hit ratio, like a trace with bursts, or skewed traffic should still offer a good effectiveness ratio, assuming that a suitable cache algorithm exists, hopefully close to OPT performance.

5. Discussion of Assumptions and Opportunities

Our empirical results indicate that our approach has potential to improve the performance of demand-aware networks. However, our study is still preliminary: on the one hand, our evaluation relies on several simplifying assumptions whose implications need to be explored further; on the other hand, we have so far focused on most basic algorithms, and we believe that our perspective opens several optimization opportunities to improve performance further. In the following, we discuss some of these assumptions and opportunities in more details.

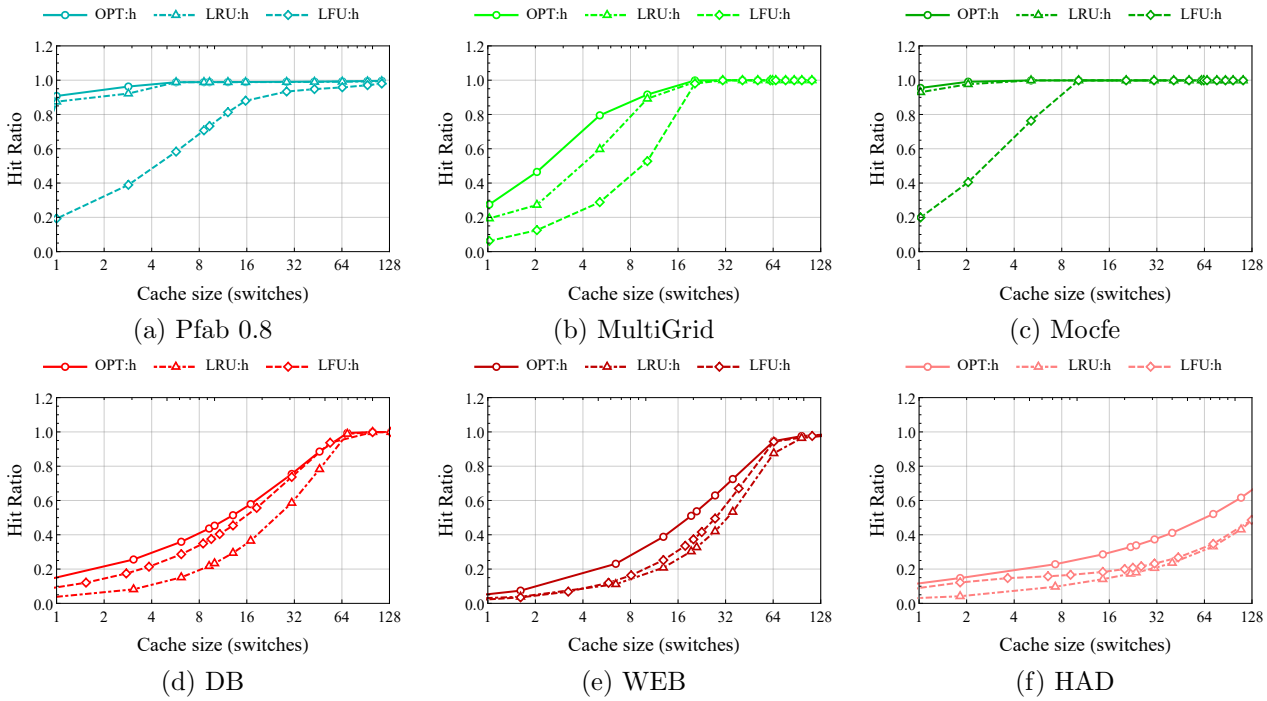


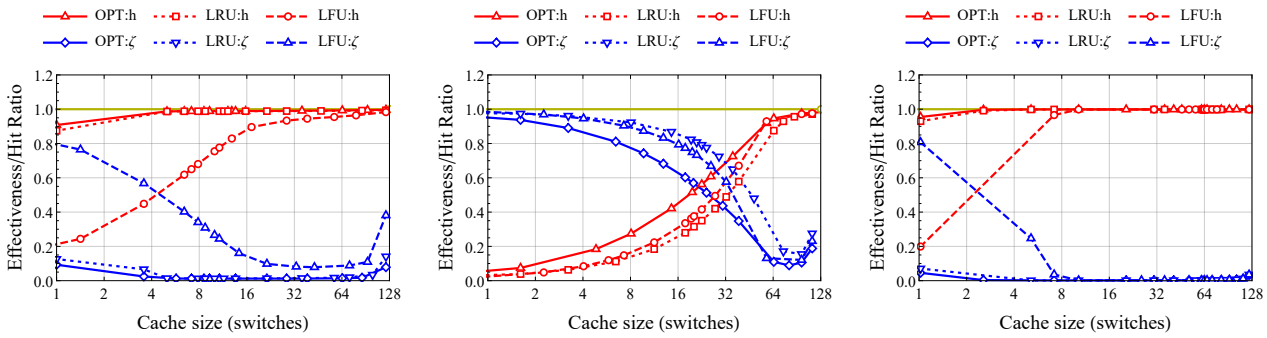
Figure 8: Hit ratio results for six different traces, using three caching algorithms, OPT, LRU, LFU.

Table 2: Scaling of *rotor-net* to our traces

Trace	FB-DB	FB-WEB	FB-HAD	HPC	pFabric	*RotorNet*
Nodes	155	325	365	1024	144	2048
Switches	128	128	128	128	128	128
Ports/switch	155	325	365	1024	144	2048
Rounds	1.21	2.56	2.85	8	1.125	16

5.1. On the Potential of Online Caching

Our empirical results suggest that for all considered traces, there is at least some potential for improvement (shown by a effectiveness ratio lower than 1), while for most traces we actually see a significant potential when employing our approach. While this may be an optimistic view on what we can achieve with our approach, given the simplifying assumptions made (see also the more detailed discussion below), our evaluation at least shows that there is structure in the communication traffic which can be captured with caching, and that the resulting performance will depend on the specific caching strategy used. For the Facebook traces in Figures 5 (a) and (b) we see that LFU is more effective than LRU, while for the HPC and pFabric traces in Figures 5 we see that LRU had the upper hand, yielding very high hit ratios. In other words, online algorithms can indeed take advantage of traffic structure and there is potential in augmenting the *links-cache* component of *CacheNet* with a clever caching strategy.



(a) Pfab 0.8 (b) WEB (c) Mocfe

Figure 9: The results of the effectiveness ratio and the hit ratio on more traces.

5.2. Scaling of Rotor-Net

One problem we still face regards the question of how to scale *rotor-net* to arbitrarily sized networks. The original RotorNet paper [13] does not discuss this aspect but relies on a rigid set of switches with a certain number of ports. However, there are several options to scale *rotor-net*. We propose to use the following rule: We begin with the baseline example from the original paper, a network with $n = 2048$ nodes (ToR) and 128 switches $k = 128$, each with 2048 in-out ports, and a single matching of size n at any slot time. This allows *rotor-net* to complete a cover of all matchings within at most 16 matchings per switch and an average cycle time of $c = \tau \frac{n-1}{k}$ (recall that τ is the *slice* time) and $m_r = nk$). We denote $\frac{n-1}{k}$ as the (average) rounds in a cycle time. We will assume that the number of switches is constant at $k = 128$. Following this rule, a network with $n = 1024$ nodes uses 128 switches each with 1024 ports, which allows us to cover all links within 8 rounds. We note that using this scaling scheme, all networks have the same number of switches, however, each switch scales with the network, with smaller networks having smaller switches. This rule can be applied to any network with a particular n. Table 2 shows the result when applied to the traces we used in this work; it shows the number of nodes, *rotor-net* switches, ports in each switch, and *average* number of rounds to cover all links using these switches. Note that the number of rounds is not necessarily an integer: it represents the *average* value.

5.3. Dealing with Higher Reconfiguration Times

While conceptually our approach is more general, in our evaluation, we made the simplifying assumption that the *links-cache* can be updated upon each packet. Although technology is improving fast [23], this is not feasible yet as reconfiguration times are still too high. However, we believe that even with a slower reconfiguration time and/or accounting only for a sample of all messages, the *links-cache* component could still profit from a smart caching and prediction algorithm, albeit with a naturally lower hit ratio. To this end, it may be also interesting to consider more general notions of requests (consisting of certain amounts of

packets) and studying a combination of both LRU and LFU, where some part of the *links-cache* functions with different policies like in [24].

More specifically, recall that we defined R_c to be the reconfiguration time of an edge in the *links-cache*, and R_r is the reconfiguration time of an edge in *rotor-net*. While R_c has not been used directly in our models' main results, it remains an important consideration when designing *CacheNet*. Let us consider how the reconfiguration times relate to the performance of *CacheNet*, and in particular how R_r relates to R_c. In our model the values of R_c and R_r are any positive real number; practically, their values depend on the available switching technology.

The main advantage of the *rotor-net* system is the low reconfiguration time which is a result of, mainly, the use of a predetermined set of possible matchings per each switch. This apparent lack of flexibility results in reconfiguration times of about $R_r = 20\mu s$. This is much faster than an all-to-all dynamic MEMS switch which can have a reconfiguration time of an order of magnitude greater, in the range of $R_c = 15 - 20ms$ [25] (which is typical of such optical switches [15]).

While it may seem that in this work we use $R_c = 0$, this is inaccurate. In a system where $R_c = 0$ all packets could be sent using a direct link, while we wait at least for the next packet to arrive for the reconfiguration to take effect; otherwise packets are routed using *rotor-net*. We believe this rule emulates a case where changes are made not for each packet, which is unrealistic, but for example, for each large flow. The actual transmission time of a large flow is far longer then the reconfiguration time of $R_c = 20ms$, meaning that until the next flow arrives, the network would have already reconfigured to include a direct link for it. Furthermore, trying to account for the reconfiguration time of real world optical switches in our simulations would be futile, as the traces are captured and sampled at a resolution of $1sec$. In a more practical implementation of *CacheNet* we hope to use switches such as the MEMS switch to realize our *links-cache*. In the simplest manner, whenever an edge is removed and a new is added to the *links-cache* the switch will change one of its connections. The longer reconfiguration times could be dealt by using a less strict cache policy. Edges don't have to necessarily change for each cache miss (or hit), they could, for example, change every constant number of misses, or only change after R_c period of time. This will allow the system to handle slow reconfiguration speeds perhaps at cost of performance. We leave the exact details of the impact of a slower cache to future work.

5.4. Simplifying Assumptions in our Analysis

For our analysis, we made several simplifying assumptions. While we understand our analysis as a first step and we leave a more general analysis for future work, we discuss some of the main limitations in the following.

Throughput: It would be nice to analytically evaluate throughput. However, while the relation between the average packet delay and throughput is not tight, lower values of the former should, in general, also

imply better throughput. In particular, *CacheNet* aims to transmit more traffic via direct links, which not only lowers the average path length between active nodes but also decreases delays caused due to congestion and queues. Since in *CacheNet* all packets are sent using a single hop, *CacheNet* has a very high bandwidth efficiency. In *CacheNet* we could lower latency by trading bandwidth efficiency in two places. First, at the *links-cache*, we could use multi-hop routing on the network induced from *links-cache* itself. Second, at the *rotor-net* element we could modify it to use multi-hop routing much like in the original Opera and RotorNet papers. We explore these ideas further in this section.

Multi-hop routing: *RotorNet* uses two-hop routing for load balancing [13]. We assume no blocking occurs due to full queues and that no two hop routing, where a packet is sent immediately over two hops on the current matching, is possible. Under these assumption and model a scheme that uses two two-hop routing would only increase the average delay per packet as all packets are already sent immediately when the correct matching is present, sending the packet to another node would only increase delay, if we have to wait for the next matching.. We therefore modeled *rotor-net* as a system where packets are transmitted along single-hop paths only. Future iterations of *CacheNet* that would also model congestion and throughput, may use multi-hop routing in the *links-cache* or rotor elements. Nodes which are not directly connected could be connected through a two hop path. This will allow *CacheNet* to send traffic along short paths, without a reconfiguration penalty and without the latency involved in using the *rotor-net* system. This might require us to update routing table partially after each change to *links-cache*, or use preconfigured routing tables for the *rotor-net* element.

Decentralized control: We believe the the concept of a *links-cache* naturally lends itself for a more decentralized control. It is possible to allow each ToR switch to choose which edge to remove from, and add into, its own *links-cache* using only locally gathered statistics. Based on these statistics switches may locally coordinate which mutual links to establish based on different metrics, much like in this paper where we consider recency and frequency. While in our simulations, we still leverage centralized information (which gives an upper bound on the potential), it will be interesting to fully make use of decentralization.

Queuing delays: Finally, our current model does not take into account the queuing delays due to either congestion or reconfiguration time. Such consideration would be very important for *CacheNet* when it progress towards a more complete model.

5.5. Other caching algorithms

In this work we have only explored three caching algorithms, only two of which, LRU and LFU are truly online. What are the potentials of other different algorithms? How significantly can we improve the performance of *CacheNet*? In fact, despite being a frailly simple algorithm it is well known that LRU has good performance grantees. LRU is worse than the optimal algorithm by no more than a factor of K, where K is the cache size. In fact no other online paging (caching) algorithm can achieve a better competitive ratio

than K [26, 27]. Of course, this refers to the worst possible case and does not mean there is no room for improvement. Our addition of the OPT algorithm was meant to give an upper bound on the best possible performance on any algorithm. Indeed, there were some cases shown in our evaluation where LRU was very different from OPT in terms of performance. In the future we hope to utilize more advance caching algorithms. An interesting avenue for research could be the development of caching algorithms with an AI predictor component, some similar work, which tries to use a neural network to reduce prediction errors are already being developed [28]. In our setting an AI algorithm might work using some reinforcements learning technique, where the algorithms learns which edges are the worst to evict in an online manner, perhaps similarly to DeepConf [29]. The AI algorithm will only evict edges which are the 'least bad'. However, we would also point out that LRU and LFU have the benefit of being simple and quick, this might not be true for more complex algorithms. This parameter could be important in the setting of communication networks, where requests need to be served quickly.

5.6. Using Opera

In this paper we use a network based upon RotorNet. Recently a more advanced version of RotorNet has been released, Opera[16]. One of the main differences between RotorNet and Opera is the use of expanders to send delay-sensitive small flows over several hops. Our work explores the average delay per packet and so we do not deal with flows explicitly, nor do we explicitly model multi-hop routing. However, we can still try and consider how would the results change (if at all) if we used Opera instead of RotorNet. Opera sends a portions of its traffic on an expander, which exists at any point as a union of matchings, with no further delay other than the transmission time. If we continue with our assumption that the transmission time is essentially zero we can conclude that the average delay of a system similar to *CacheNet* that uses Opera instated of *RotorNet* will have an average packet delay t_o that is a factor β smaller than the delay of our system t_r that is $t_o = \beta t_r$, where β is the fraction of packets which belong to small flows. Assuming that all packets are of the same size, β would equal the amount of bytes which belong to small flows. Looking at the Opera paper we see that β should typically be close to 1, therefore the change should be minor. But what would change if β is significantly smaller than 1? We can calculate the effectiveness ratio again for Opera such as in Equation 1, but fully replacing *rotor-net* with Opera. Recall that we assume that the transmission time t tend to zero.

$$\zeta_m(m_r, h) = \frac{ht_c + (1-h)t_o}{t_o^*} = \frac{ht_c + \beta(1-h)t_r}{\beta t_r^*} = \frac{ht + \beta(1-h)\frac{\tau}{2}\frac{n(n-1)}{m_r} + t}{\beta\frac{\tau}{2}\frac{n(n-1)}{m} + t} = (1-h)\frac{m}{m_r} \tag{8}$$

This means that when measuring the effectiveness ratio, β will cancel out, leaving us with the same results. To clarify, we point out that the overall performance of *CacheNet* will improve if it uses Opera, however, the effectiveness ratio remains the same.

6. Related Work

CacheNet builds upon several innovative technologies recently developed to improve datacenter networks. We will organize our review of related works according to the types considered in this paper: static and dynamic, where the latter is further subdivided into demand-oblivious (such as *rotor-net*) and demand-aware (such as cache-net). Most existing datacenter designs rely on static topologies. The Clos topologies and multi-rooted fat-trees are the most widely deployed datacenter networks, and come in different flavors [30, 31, 32]. Recently, modular hypercubic networks have received much attention [33, 34] as well as expander-based networks [35, 36]. An innovative alternative to static topologies has been introduced by Mellette et al. In their first work, RotorNet [13], a scalable optical datacenter network design (circuit-based), the authors show that very high bandwidth can be provided by actually emulating a full-mesh network, dynamically reconfiguring the circuit switches constituting the datacenter. RotorNet is a hybrid design and serves low-latency traffic over a static network. In a follow up work, Opera [16], the authors improve upon RotorNet by presenting a rapid and deterministic reconfiguration scheme which ensures that at any moment in time, the network implements an expander graph while over time, bandwidth-efficient single-hop paths are provided between all racks. The networks discussed above have in common that their topology does not depend on the current demand. In contrast, the goal of demand-aware networks is to exploit specific structure in the workload, which is motivated by several interesting measurement studies. Indeed, traffic matrices are known to be sparse and skewed [10, 37, 8], and traffic can be bursty over time [38, 39]. Recently, Avin et al. [10] presented a methodology to measure the spatial and temporal locality of datacenter traffic, Other existing empirical studies all confirm that traffic patterns in datacenters are often *sparse and skewed* [40, 8]. They hence should contain locality which cache algorithms can exploit.

Existing demand-aware networks can be classified according to the granularity of reconfigurations. Solutions such as OSA [41] or DANs [42], among other, are more coarse-granular and e.g., rely on a (predicted) traffic matrix. Solutions such as ProjecToR [40], MegaSwitch [43], Eclipse [3], Helios [2], Mordia [7], C-Through [44] or SplayNets [45] as can be seen in the survey [15] are more fine-granular and support per-flow reconfiguration and decentralized reconfigurations. Due to the increased reconfiguration time experienced in demand-aware networks, many of these solutions additionally rely on a fixed network. For example, ProjecToR always maintains a "base mesh" of connected links that can handle low-latency traffic while it opportunistically reconfigures free-space links in response to changes in traffic patterns. To given another example, OSA allows to reserve some circuit-switch ports specifically to ensure connectivity for low-latency traffic. MegaSwitch could support low-latency traffic in a similar manner. While our work tries to use simple caching algorithms to optimize the network towards network traffic, other works such as xWeaver[46] and DeepConf [29] use various deep learning methods to study the traffic pattern in data center networks

The framework and model of this work are inspired from [47] where the authors gave a general model for

demand-aware networks. However, the current work takes one step towards more concrete research, offering architecture and algorithms and using real data to evaluate the demand-aware network.

However, we are not aware of any work exploring the opportunities of a distributed *links-cache* to enhance an otherwise demand-oblivious topology. Furthermore, while prior work primarily relied on ad-hoc methodologies, we are not aware on any formal and unifying model which allows us to analytically quantify to which extent demand-aware cache links are optimally combined with demand-oblivious links.

7. Conclusion

In order to efficiently leverage locality in reconfigurable datacenter networks, we proposed to extend a demand-oblivious network topology with a distributed *links-cache*. Our solution, *CacheNet*, also allows us to study how to optimally partition a network into demand-oblivious and demand-aware topology components, supporting both analytical and empirical evaluations. We understand our work as a first step and believe that it opens several interesting avenues for future research. In particular, it will be interesting to explore additional and alternative link caching algorithms, and extend our model to include additional types of links, such as static links. More generally, it will be interesting to explore the implications of reconfigurable optical networks on other layers of the network stack, in particular, on routing and congestion control.

Acknowledgement: This project received funding by the European Research Council (ERC), grant agreement no. 864228, Horizon 2020, 2020-2025.

References

[1] Chen Griner and Chen Avin. Cachenet: Leveraging the principle of locality in reconfigurable network design. *IFIP Networking conference*, 2021.

[2] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM CCR*, 41(4):339–350, 2011.

[3] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. *Queueing Systems*, 88(3-4):311–347, 2018.

[4] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *ACM SIGMETRICS*, volume 44, pages 75–88, 2016.

[5] He Liu, Matthew K Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M Voelker, David G Andersen, Michael Kaminsky, et al. Scheduling techniques for hybrid circuit/packet networks. In *Proc. of the ACM ConNEXT*, page 41, 2015.

[6] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *USENIX NSDI*, pages 1–15, USA, 2014.

[7] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *ACM SIGCOMM CCR*, volume 43, pages 447–458, 2013.

[8] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM CCR*, volume 45, pages 123–137, 2015.

[9] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proc. 1st ACM Workshop on Research on Enterprise Networking (WREN)*, pages 65–72. ACM, 2009.

[10] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–29, 2020.

[11] Chen Avin, Chen Griner, Iosif Salem, and Stefan Schmid. An online matching model for self-adjusting tor-to-tor networks. *arXiv:2006.11148*, 2020.

[12] Peter J. Denning and Peter J. The locality principle. *Communications of the ACM*, 48(7):19, jul 2005.

[13] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proc. of ACM SIGCOMM*, pages 267–280, 2017.

[14] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. ACM CoNEXT*, pages 205–219, 2016.

[15] Klaus-Tycho Foerster and Stefan Schmid. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *SIGACT News*, 50(2):62–79, July 2019.

[16] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. *arXiv preprint arXiv:1903.12307*, 2019.

[17] US DOE. Characterization of the DOE mini-apps. `https://portal.nersc.gov/project/CAL/doe-miniapps.htm`, 2016.

[18] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM CCR*, volume 43, pages 435–446, 2013.

[19] Ke Wen, Payman Samadi, Sébastien Rumley, Christine P Chen, Yiwen Shen, Meisam Bahadori, Keren Bergman, and Jeremiah Wilke. Flexfly: Enabling a reconfigurable dragonfly through silicon photonics. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 166–177. IEEE, 2016.

[20] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[21] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[22] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[23] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proc. ACM SIGCOMM*, pages 782–797, 2020.

[24] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[25] MEMS-Optical-Switches. `http://www.diconfiber.com/products/mems_matrix_optical_switches.php`.

[26] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[27] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[28] Vincenzo Eramo, Francesco Giacinto Lavacca, Tiziana Catena, and Paul Jaime Perez Salazar. Application of a long short term memory neural predictor with asymmetric loss function for the resource allocation in nfv network architectures. *Computer Networks*, 193:108104, 2021.

[29] Saim Salman, Christopher Streiffer, Huan Chen, Theophilus Benson, and Asim Kadav. Deepconf: Automating data center network topologies management with machine learning. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 8–14, 2018.

[30] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[31] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[32] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 399–412, 2013.

[33] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[34] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. Mdcube: a high performance network structure for modular data center interconnection. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 25–36. ACM, 2009.

[35] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proc. ACM SIGCOMM*, pages 281–294. ACM, 2017.

[36] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers, randomly. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 12, pages 17–17, 2012.

[37] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[38] Shihong Zou, Xitao Wen, Kai Chen, Shan Huang, Yan Chen, Yongqiang Liu, Yong Xia, and Chengchen Hu. Virtualknotter: Online virtual machine shuffling for congestion resolving in virtualized datacenter. *Computer Networks*, 67:141–153, 2014.

[39] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM.

[40] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229. ACM, 2016.

[41] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. Osa: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking*, 22(2):498–511, April 2014.

[42] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. In *Proc. International Symposium on Distributed Computing (DISC)*, 2017.

[43] Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling wide-spread communications on optical fabric with megaswitch. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 577–593, USA, 2017. USENIX Association.

[44] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. *ACM SIGCOMM Computer Communication Review*, 41(4):327–338, 2011.

[45] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking (ToN)*, 24(3):1421–1433, 2016.

[46] Mowei Wang, Yong Cui, Shihan Xiao, Xin Wang, Dan Yang, Kai Chen, and Jun Zhu. Neural network meets dcn: Traffic-driven topology adaptation with deep learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):26, 2018.

[47] Chen Avin and Stefan Schmid. Toward demand-aware networking: A theory for self-adjusting networks. *ACM SIGCOMM Computer Communication Review*, 48(5):31–40, 2019.