Routing Algorithms for Recursively-Defined Datacenter Networks

Alejandro Erickson[1], Javier Navaridas[2], Jose A. Pascual[2], and Iain A. Stewart[1]

[1]*School of Engineering and Computing Sciences, Durham University, South Road, Durham DH1 3LE, U.K.*
[2]*School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.*

Abstract

The server-centric datacenter network architecture can accommodate a wide variety of network topologies. Newly proposed topologies in this arena often require several rounds of analysis and experimentation in order that they might achieve their full potential as datacenter networks. We propose a family of novel routing algorithms for completely-connected recursively-defined datacenter networks that are immediately applicable to the well-known datacenter networks (Generalized) DCell and FiConn. Our routing algorithms involve the concept of proxy search and we show, via an extensive experimental evaluation, that our methodology leads to significant improvements in terms of hop-length, fault-tolerance, load-balance, network-throughput, end-to-end latency, and workload completion time when we compare our algorithms with the existing routing algorithms for (Generalized) DCell and FiConn.

1 Introduction

The explosive growth of online services powered by datacenters (web search, cloud computing, *etc.*) has motivated intense research into datacenter network (DCN) design over the past decade and brought about major breakthroughs. For example, fat-tree DCNs, introduced in [2], use commodity off-the-shelf (COTS) servers and switches in a fat-tree (topology), and have resulted in an evolutionary shift in production datacenters towards leaf-spine topologies. Fat-tree DCNs are not a panacea, however, as they suffer in terms of scalability [17] and flexibility [21].

Research on DCN architecture is ongoing and each new architecture invites the use of new classes of topologies. Fat-trees are the prevailing example of indirect networks, where servers are the terminals connected to a switching fabric. However, a host of alternative topologies can be implemented as indirect networks, including random regular graphs ([21]) and butterfly networks ([14]). Alternatively, the DCN Helios ([10]) is an example of a hybrid electrical/optical switch architecture and has the capacity to accommodate a variety of topologies (both in the wired links as well as in the optical switch itself). Each architecture sets constraints on the topology in a variety of ways; for example, by the separation of switch-nodes from server-nodes or the number of ports in the available hardware.

The server-centric DCN (SCDCN) architecture, introduced in [12], is such that only dumb crossbar-like switches are used, with the responsibility for routing packets within the network residing with the servers. The switches have no knowledge of the network topology and are thus only connected to servers; servers, on the other hand, may be connected to both switches and servers. The SCDCN paradigm has resulted in a variety of new DCN designs, derived both from existing and well-understood topologies in interconnection networks and as original topologies geared explicitly towards DCNs (*e.g.*, [1,11–13,17,18]).

In this paper, we are concerned primarily with routing algorithms for the two well-known SCDCNs DCell ([12]) and FiConn ([17]), and a variation of the former called β-DCell (from a family of extensions of DCell called Generalized DCell, developed in [15,16][1]). Our research involves the development of novel, generic routing algorithms, applicable to the DCNs mentioned above, along with an extensive empirical analysis.

We characterise (Generalized) DCell and FiConn as special cases of completely connected recursively-

[1]The routing algorithms and evaluation in this paper are an extension of preliminary ideas presented at the International Symposium on Parallel and Distributed Processing [9].

defined networks (CCRDNs), *i.e.*, each one is a parameterised family of networks such that a level-k member of the family is constructed from a number of disjoint level-$(k-1)$ members that are all pairwise linked together by additional level-k links. Using the CCRDN classification we, first, unify the basic traffic oblivious routing algorithms defined for DCell, Generalized DCell, and FiConn in [12], [15,16], and [17], respectively, namely `DCellRouting`in DCell and Generalized DCell and `TOR`in FiConn, as the canonical routing algorithm `DR`, and second, develop a parameterised classification (which, to our knowledge, is new) of all possible routes between endpoints of the DCNs (Generalized) DCell and FiConn (below, we use k to denote the level, or recursive depth, of some CCRDN, with n denoting the radix of the embedded switch-nodes).

Next, we develop a novel family of routing algorithms called `PR` (or ProxyRoute) from within our classification. In brief, given server-nodes *src* and *dst* in distinct level-$(k-1)$ substructures (within the same level-k structure), `PR` employs a subroutine called `GP` in order to select a third level-$(k-1)$ substructure (not containing *src* or *dst*), called a proxy, through which a route is constructed. Considering and comparing potential proxies is the computational bottleneck of `PR`. Thus, `PR`'s main feature is that it leverages the topological structure of CCRDNs in a widely applicable way so that the selection routine performs an intelligent and reduced search of potential proxies, with the aim being to discover 'good' proxies at low computational cost. Our intelligent proxy selection routines are called `GP_I` and `GP_O` and, as a baseline reference, we also evaluate a brute-force search of all potential proxies, called `GP_E`.

We evaluate the performance of `PR`, with each of its subroutines `GP_E`, `GP_I`, and `GP_O`, by comparing `PR` with the different versions of `DR` in DCell, β-DCell, and FiConn, as well as with the existing traffic-aware and fault-tolerant routing algorithms `DFR` for DCell ([12]) and `TAR` for FiConn ([17]), and a breadth-first search routing algorithm (as a guiding yard-stick). We explore a comprehensive set of metrics using the flow-based DCN simulator `INRFlow` [8], and evaluate `PR` and the other routing algorithms on (server-node-to-server-node) hop-length, fault-tolerance, load-balance, network-throughput, end-to-end latency, and workload completion time, for level $k \in \{2, 3\}$ DCNs ranging from 15 thousand to over 3 million server-nodes, and with a variety of workloads that are typical in DCN usage.

Our results are extremely encouraging indeed. For many of the topologies we consider, `PR` performs very well with respect to all of the aforementioned metrics, and in most aspects far exceeds the performance of existing routing algorithms. For example, in a fault-free β-DCell, `PR` with `GP_I` computes paths that are up to 16% shorter (as regards hop-length) than `DR`, and when 10% of the links are made faulty (uniformly at random), `PR` provides 85% connectively and still retains the same mean hop-length as `DR` does in a fault-free environment (note that there is no existing fault-tolerant routing algorithm for β-DCell). As another example, in terms of connectivity in DCell when 10% of the links are faulty, `PR` with `GP_I` has similar performance to the existing fault-tolerant routing algorithm `DFR` but `DFR` computes paths that are over 35% longer than those computed by `PR` with `GP_I`. `PR`'s propensity for finding different available paths through its search for proxies translates into good load balancing properties, and consequently we see an improvement in throughput; for example, when comparing `PR` with `GP_I` in β-DCell against `DR`, we obtain a dramatic 55% improvement in throughput. Moreover, the intelligent, reduced search for proxies, coupled with shorter hop-lengths, allows these performance gains to come at no cost (and in certain cases, some improvement) in latency. Finally, the improvements in hop-length, load-balance, throughput, and latency are integrated into a comprehensive, dynamic workload execution experiment using `INRFlow`. We find that the gains observed in each aspect above translate so that using `PR` can reduce the completion times of our benchmark workloads by up to 40%.

The improvements obtained by `PR` vary from one topology to another, including according to the connection rules of DCell and β-DCell (with the same level parameter k). More significant gains are made in level 3 DCNs than in level 2, and for $k = 3$ we find greater performance gains in low radix (n) DCNs than in high radix ones. The gains for level 3 DCells are significant, but they trail those in β-DCell; for example, the mean hop-length achieved by `PR` in DCell is closer to 10% better than that of `DR`, and the connectivity when 10% of the links are faulty is closer to 80% (see above for the situation for β-DCell). In so far as we are aware, our results are the first to highlight that the actual connection rule used in DCell can substantially affect performance. In contrast, the gains achieved by `PR` over `DR` in FiConn are limited to hop-length and the throughput of very light workloads. Our analysis provides insight to these

limitations.

The remainder of the paper is organised as follows. We give essential definitions in Sections 2 and 3, where we abstract the DCNs (Generalized) DCell and FiConn as graphs which can be characterised as CCRDNs. Section 4 describes previously known routing algorithms for these DCNs, in the context of CCRDNs, and our classification of routes in CCRDNs is given in Section 5. We continue Section 5 with the presentation of our main contribution, namely the design of `PR`, and Sections 6 and 7 describe our methodology and evaluation of `PR`, respectively. We conclude in Section 8 by identifying future avenues for research.

2 Server-centric DCNs

Our results and experiments involve graph-theoretic abstractions of SCDCNs; therefore, it is appropriate that we define these abstractions precisely.

An SCDCN consists of servers and switches, which act only as crossbars and have no routing intelligence. These components are linked together, with the only restriction being that a switch cannot be linked directly to another switch (we assume all links are bidirectional). As such, an SCDCN is abstracted here by an undirected graph $G = (W \cup S, E)$, with the set of nodes partitioned into the subset of *switch-nodes*, W, and the subset of *server-nodes*, S. Naturally, each switch of the SCDCN corresponds to a switch-node of W, and each server corresponds to a server-node of S. Each link of the SCDCN corresponds to an edge e of E, which, for convenience, we shall also call a link. The condition that switch-to-switch links are not allowed in SCDCNs implies that $E \cap (W \times W) = \varnothing$. (Throughout, we refer the reader to [6] for undefined graph-theoretic terms.)

Also relevant to our discussion of routing algorithms in SCDCNs is the fact that (1) packets are sent and received only by servers, and (2) packets undergo a negligible amount of processing time in each switch, compared to the time spent in each server. The reason for (2) is that the SCDCN paradigm requires that packets are routed only in the server; therefore they have to travel through the protocol stack (up and down) to reach the service that will perform the routing. As a result it is reasonable to assume that packets spend much more time traversing a server than a switch.

The outcome of (1) is that we need only discuss routing algorithms that construct paths whose endpoints are server-nodes; that is, a *route* in an SCDCN is a path whose endpoints are server-nodes and whose intermediate nodes can be switch-nodes or server-nodes. The outcome of (2) is that, for the purposes of estimating latency, a *hop* from a server-node to a server-node is indistinguishable from one that also passes through an interim switch-node; indeed, henceforth we refer to a path consisting of a server-node followed by a switch-node followed by a server-node as a hop.

3 Recursively-Defined Networks

We are concerned with network topologies of a certain structure that have arisen frequently in the area of interconnection networks, and also recently as SCDCNs.

Definition 3.1. *A family $\mathcal{X} = \{X(h) : h = 0, 1, \ldots\}$ of interconnection networks is* recursively-defined *if each $X(h)$, where $h > 0$, is the disjoint union of copies of $X(h-1)$ together with the addition of extra links joining nodes in the different copies; in such a case, we say that $\mathcal{X}$ is an RDN. An RDN $\mathcal{X}$ is* completely-connected *(see, e.g., [4]) if there is at least one link joining every copy of $X(h-1)$ within $X(h)$ to every other copy; in such a case, we say that $\mathcal{X}$ is a CCRDN.*

Henceforth, we usually refer to both a family of interconnection networks and a particular (parameterized) member of the family as an interconnection network (moreover, we do this with DCNs). This causes no confusion.

We now give definitions of the established SCDCNs of interest to us in this paper. While the definitions are brief, they are complete and suffice to implement the SCDCNs; for additional details, we refer the reader to the requisite papers.

3.1 The DCN DCell

The first SCDCN proposed was DCell ([12]). DCell is a CCRDN, and it is defined as follows.

Fix some $n > 2$. $\mathrm{DCell}_{0,n}$ consists of one switch-node connected to n server-nodes. For $k \geqslant 0$, let t_k be the number of server-nodes in $\mathrm{DCell}_{k,n}$. For $k > 0$, $\mathrm{DCell}_{k,n}$ consists of $t_{k-1} + 1$ disjoint copies of $\mathrm{DCell}_{k-1,n}$, named D^i_{k-1}, for $0 \leqslant i \leqslant t_{k-1}$. Each pair of distinct $\mathrm{DCell}_{k-1,n}$s is joined by exactly one

link, called a *level-k link*, whose precise definition is as follows.

Label a server-node of $\text{DCell}_{k,n}$, for some $k > 0$, by $x = x_k x_{k-1} \cdots x_0$, where $x_{k-1}x_{k-2}\cdots x_0$ is the (recursively-defined) label of a server-node in $D_{k-1}^{x_k}$, and $0 \leqslant x_0 < n$ and $0 \leqslant x_i \leqslant t_{i-1}$, for $i > 0$. The labels of the server-nodes of $\text{DCell}_{k,n}$ are mapped bijectively to the set $\{0, 1, \ldots, t_k - 1\}$ by $uid_k(x) = x_k t_{k-1} + x_{k-1}t_{k-2} + \cdots + x_1 t_0 + x_0$, and we say that $uid_k(x)$ is the *uid* of the server-node with label x.

Let $0 \leqslant x_k < y_k \leqslant t_{k-1}$ be the indices of the $\text{DCell}_{k-1,n}$s named $D_{k-1}^{x_k}$ and $D_{k-1}^{y_k}$. A level-k link connects the server-node with uid $y_k - 1$ in $D_{k-1}^{x_k}$ to the server-node with uid x_k in $D_{k-1}^{y_k}$.

3.1.1 Generalized DCell

The definition of the DCN DCell generalises readily; see [15,16]. The key observation is that the level-k links are a perfect matching of the server-nodes in the disjoint copies of the $\text{DCell}_{k-1,n}$s, where every pair of distinct $\text{DCell}_{k-1,n}$s is joined by a link. Many such matchings are possible, and any such matching ρ_k defines the level-k links and is called the *ρ_k-connection rule* ([16]).

A *Generalized $DCell_{k,n}$* inherits the definition of $\text{DCell}_{k,n}$, for $k \geqslant 0$, except that the level-k links may satisfy an arbitrary connection rule. Note that a given family of Generalized DCells can be specified by a set of connection rules $\{\rho_1, \rho_2, \rho_3, \ldots\}$, with the level-$k$ links given by the ρ_k-connection rule. This is in accordance with Definition 1 in [16], with two exceptions: we model Generalized $\text{DCell}_{0,n}$ as a switch-node connected to n server-nodes, rather than as a clique of n server-nodes (this allows us to model congestion and faults in individual links); and we require that $n > 2$.

In order to demonstrate the impact of different connection rules on the routing algorithms presented in Section 4, we consider a different connection rule to the one for DCell, above. For this purpose, we study β-DCell, defined by the β-connection rule given in [16] as follows.

We refer to the disjoint copies of β-$\text{DCell}_{k-1,n}$ within any β-$\text{DCell}_{k,n}$ using the names $B_{k-1}^{x_k}$, for $0 \leqslant x_k \leqslant t_{k-1}$. Let $0 \leqslant x_k < y_k \leqslant t_{k-1}$ be the indices of the β-$\text{DCell}_{k-1,n}$s named $B_{k-1}^{x_k}$ and $B_{k-1}^{y_k}$. A level-k link connects the server-node with uid $y_k - x_k - 1$ in $B_{k-1}^{x_k}$ to the server-node with uid $t_{k-1} - y_k + x_k$ in $B_{k-1}^{y_k}$.

3.2 The DCN FiConn

One of the issues with (Generalized) $\text{DCell}_{k,n}$ is that each server-node has degree $k+1$. This requires that each server have $k+1$ NIC ports, which is not typically the case for COTS servers when $k > 1$, and thus motivates research on a special class of SCDCNs with a *dual-port* property. FiConn, proposed in [17], has the dual-port property; it is a CCRDN in which the server-nodes are of degree at most two. The definition of FiConn is similar to, but slightly different from, those of DCell and Generalized DCell, as we now see.

Fix some even $n > 3$. $\text{FiConn}_{0,n}$ consists of one switch-node connected to n server-nodes. Let b_{k-1} be the number of available server-nodes in $\text{FiConn}_{k-1,n}$, for $k > 0$, where a server-node is said to be *available* in $\text{FiConn}_{k-1,n}$ if it has degree one (note that it will always be the case that b_{k-1} is even). Build $\text{FiConn}_{k,n}$ from $b_{k-1}/2 + 1$ copies of $\text{FiConn}_{k-1,n}$, named F_{k-1}^i, for $0 \leqslant i \leqslant b_{k-1}/2$. Each pair of distinct $\text{FiConn}_{k-1,n}$ s is joined by exactly one link, again called a *level-k link*, whose precise definition is as follows (note that after we have added level-k links, we still have server-nodes of degree one).

From [17] we have that $b_{k-1}/2 + 1 = t_{k-1}/2^k + 1$, where t_{k-1} is the total number of server-nodes in $\text{FiConn}_{k-1,n}$. Label a server-node of $\text{FiConn}_{k,n}$ by $x = x_k x_{k-1} \cdots x_0$, where $x_{k-1}x_{k-2}\cdots x_0$ is the (recursively-defined) label of a server-node in $F_{k-1}^{x_k}$, and $0 \leqslant x_0 < n$ and $0 \leqslant x_i \leqslant t_{i-1}/2^i$, for $i > 0$. As before, we have $uid_k(x) = x_k t_{k-1} + x_{k-1}t_{k-2} + \cdots + x_1 t_0 + x_0$ and so we obtain a bijection from the server-nodes of $\text{FiConn}_{k,n}$ to the set $\{0, 1, \ldots, t_k - 1\}$.

Let $0 \leqslant x_k < y_k \leqslant t_{k-1}/2^k$ be the indices of the $\text{FiConn}_{k-1,n}$s $F_{k-1}^{x_k}$ and $F_{k-1}^{y_k}$. A level-k link connects server-node $(y_k - 1)2^k + 2^{k-1} - 1$ in $F_{k-1}^{x_k}$ to server-node $x_k 2^k + 2^{k-1} - 1$ in $F_{k-1}^{y_k}$.

4 Routing in completely-connected recursively-defined networks

Having defined the concept of a CCRDN and provided some examples thereof, we now consider how CCRDNs give rise to a certain class of routing algorithms and how such routing algorithms have been developed for DCell, Generalized DCell, and FiConn(albeit outside our encompassing framework).

4.1 Dimensional routing

CCRDNs feature a class of routing algorithms that emerges naturally from the definition of a CCRDN, called *dimensional routing*. With reference to Definition 3.1, recall that a CCRDN $X(h)$, where $h > 0$, consists of disjoint copies of $X(h-1)$ so that there is at least one 'bridging' link joining (server-nodes in) any two distinct copies of $X(h-1)$. Dimensional routing uses the recursive structure and these bridging links to build routes in a canonical fashion.

Definition 4.1. *Let $\mathcal{X} = \{X(h) : h = 0, 1, \ldots\}$ be a CCRDN, and let X^a_{h-1} and X^b_{h-1} be disjoint copies of $X(h-1)$ in $X(h)$, where $h > 0$. Let src and dst be server-nodes of X^a_{h-1} and X^b_{h-1}, respectively. There is a level-h link in $X(h)$ incident with a server-node dst' in X^a_{h-1} and a server-node src' in X^b_{h-1}. A path P_a from src to dst' can be recursively computed in X^a_{h-1}, as can a path P_b from src' to dst in X^b_{h-1} (we assume that the computation of paths in $X(0)$ is trivial). A* dimensional routing algorithm *on $\mathcal{X}$ is one which computes paths of the form $P_a + (dst', src') + P_b$ between any source-destination pair of server-nodes as above, and is denoted* `DR`. *A* dimensional route *is one that can be computed by a dimensional routing algorithm.*

It should be noted that although dimensional routing algorithms are natural algorithms and relatively easy to implement, it need not be the case that there exists a dimensional routing algorithm that always computes a shortest path. An example of an interconnection network for which this is the case is the WK-recursive network ([22]; a shortest path routing algorithm, that is not a dimensional routing algorithm, is developed in [7]).

Of course, there are canonical dimensional routing algorithms for DCell, Generalized DCell, and FiConn; canonical in the sense that in the completely-connected recursive decomposition for any of these DCNs, there is exactly one link joining any two sub-DCNs (and in each case the base sub-DCN $X(0)$ consists of a switch-node with adjacent server-nodes). These dimensional routing algorithms, namely `DCellRouting` for DCell and `TOR` for FiConn, form the basis for the fault-tolerant and load-balancing routing algorithms `DFR` for DCell ([12]) and `TAR` for FiConn ([17]), and the dimensional routing algorithm for Generalized DCell is precisely the algorithm called (generalized) `DCellRouting` for Generalized DCell ([16]). Just as for WK-recursive networks, these dimensional routing algorithms do not always yield shortest paths. We now briefly overview these routing algorithms (we shall return to their performance later when we compare these algorithms with our own).

4.2 Existing routing algorithms for DCell, Generalized DCell, and FiConn

4.2.1 DCell

The DCN DCell is presented in [12] with a fault-tolerant routing algorithm called `DFR` that is built upon `DCellRouting`. `DFR` computes a route in a distributed manner, making decisions on the fly, based on information that is local to the current location of the packet being routed (this strategy can clearly be used to handle both congestion and faulty links). Essentially, for much of the time `DFR` behaves as `DCellRouting` does, except that when routing within some $\text{DCell}_{l,n}$, with l small (l is user-chosen and usually $l \leqslant 2$), `DFR` circumvents faulty or congested links as follows.

Let D^a_{l-1} and D^b_{l-1} be $\text{DCell}_{l-1,n}$s for which the link (n_1, n_2) from D^a_{l-1} to D^b_{l-1} is overly congested or faulty. If (n_1, n_2) is the next intended hop then `DFR` selects D^c_{l-1}, a third $\text{DCell}_{l-1,n}$ (distinct from D^a_{l-1} and D^b_{l-1}) which is linked to a server-node p_1 in D^a_{l-1} that is 'near' n_1 (within D^a_{l-1}), by the link (p_1, p_2). The packet, now at n_1, is then re-routed to p_1 (within D^a_{l-1}) and then on to p_2, whereupon `DFR` behaves as does `DCellRouting`. The server-node p_1 or the $\text{DCell}_{l-1,n}$ D^c_{l-1} is known as a *proxy*.

The algorithm `DFR` has a facility to collect local data (within some $\text{DCell}_{l,n}$) that is used to decide when to re-route (as above) so as to avoid congested links. We omit the details, but this data collection can also be used to detect a faulty link (n_1, n_2) and begin re-routing through (p_1, p_2) before the packet actually arrives at n_1.

4.2.2 Generalized DCell

The routing algorithms developed for Generalized DCell are analogous to `DCellRouting` [16], so that dimensional routing is used for each topology. Note that no fault-tolerant routing algorithm has so far been developed for Generalized DCell.

4.2.3 FiConn

The DCN FiConn is presented in [17] with a fault-tolerant routing algorithm called TAR that is built upon the dimensional routing algorithm TOR. TAR is built around TOR but attempts to better utilize the two links incident with any server-node so as to balance loads and obtain a better overall throughput. The basic idea is as follows.

Prior to the transport of a flow, a route is set up by a path-probing packet, greedily and on a hop-by-hop basis, so that at some server-node s: if the next link to be used (according to TOR) is a level-l link (from s and where $l \geqslant 1$) then an available bandwidth comparison is undertaken on the level-0 link incident with s and with the switch-node w; if the available bandwidth of the level-0 link is greater than that of the level-l link then a randomly chosen server-node s' that is adjacent to w is chosen with the next two links on the path being (s, w) and (w, s'); otherwise, the level-l link (as recommended by TOR) from s is chosen. When the route has been constructed, the flow is transported, with a path-probing packet being periodically sent out (to keep the route 'fresh' in relation to the dynamic traffic states).

Note that the ideas inherent within TAR have some similarity with those used to develop DFR in DCell in that within $\text{FiConn}_{0,n}$, proxy server-nodes are chosen so that the path may diverge from that recommended by TOR.

5 Proxy routing

We now present a generic formulation of more sophisticated routing algorithms for CCRDNs that generalize the above routing algorithms for DCell, Generalized DCell, and FiConn. Henceforth, we assume that any CCRDN $\mathcal{X} = \{X(h) : h = 0, 1, 2, \ldots\}$ is such that within any $X(h)$, there is exactly one bridging link joining any pair of distinct copies of $X(h-1)$. Our algorithms make a more extensive use of proxies than was undertaken previously (in DFR and TAR) and we refer to the general class of algorithms that we develop as *proxy routing algorithms*, which we define as follows.

Definition 5.1. *Let $\mathcal{X} = \{X(h) : h = 0, 1, \ldots\}$ be a CCRDN. Fix $X(h)$ and let $X^0_{h-1}, X^1_{h-1}, \ldots, X^{t-1}_{h-1}$ be distinct copies of $X(h-1)$ in $X(h)$. Moreover: let P_i be a path in X^i_{h-1} from the server-node src_i to the server-node dst_i, for $0 \leqslant i \leqslant t-1$; and let (dst_i, src_{i+1}) be a level-h link, for $0 \leqslant i \leqslant t-2$. A* general route of order t *is a path of the form*

$$P_0 + (dst_0, src_1) + P_1 + \cdots + (dst_{t-2}, src_{t-1}) + P_{t-1}.$$

A proxy route *is a general route of order* 3*, and a* dimensional route *is a general route of order* 2*. Of particular interest to us will be* proxy routing algorithms *which are routing algorithms that produce proxy routes.*

A comparison of dimensional and proxy routes can be visualised in $\text{FiConn}_{2,4}$ in Fig. 1.

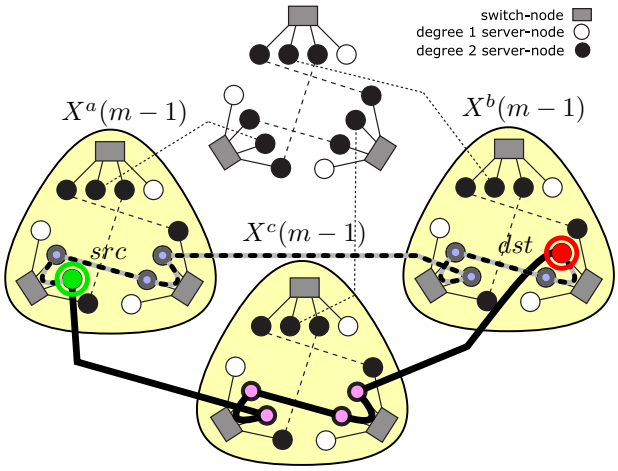


Figure 1: Dimensional route (dashed black-grey), with 7 hops, and proxy route (thick black), with 5 hops, highlighted on a $\text{FiConn}_{2,4}$, with the notation from Definition 5.1.

The proxy routing algorithms that we develop follow the spirit of DFR and TAR in that they provide some capacity for fault-tolerance and load-balancing through the (non-deterministic though guided) choice of proxies. We choose general routes of order 3 as proxy routes for two reasons: first, (with reference to Definition 5.1) re-routing through a proxy X^i_{h-1} comes with an associated computational cost and to re-route through more than one such proxy could be counter-productive; and, second, looking for 'good' proxies (as we do) can become impractical when there is more than one such proxy.

Our generic proxy routing algorithm PR for CCRDNs is detailed in Algorithm 1 (our notation is as in Definition 5.1). Intrinsic to the algorithm PR is the subroutine GP that computes the proxy used in Expression (1), if a proxy is to be used. The subroutine GP returns either a proxy X^c_{h-1} or it returns *null*. Obviously, the performance of PR (and its success in

producing a shorter route than dimensional routing) depends both on the proxy returned by GP and how GP is implemented.

Algorithm 1 PR for $\mathcal{X}$ returns a proxy route if it finds one that is shorter than the corresponding dimensional route.

Require: *src* and *dst* are server-nodes in a $X(h)$.
  function PR(src, dst, h)
    if $h > 0$ and both *src* and *dst* are in the same copy of $X(h-1)$ **then**
      return PR($src, dst, h-1$)
    end if
    $X^c_{h-1} \leftarrow$ GP(src, dst, h)
    if $X^c_{h-1} = null$ **then**
      return DR(src, dst)
    else
      $X^a_{h-1} \leftarrow$ the copy of $X(h-1)$ containing *src*
      $X^b_{h-1} \leftarrow$ the copy of $X(h-1)$ containing *dst*
      $(a^c, c^a) \leftarrow$ the link from X^a_{h-1} to X^c_{h-1}
      $(c^b, b^c) \leftarrow$ the link from X^c_{h-1} to X^b_{h-1}
      return

$$\texttt{PR}(src, a^c, h-1) + (a^c, c^a) + \texttt{PR}(c^a, c^b, h-1) + (c^b, b^c) + \texttt{PR}(b^c, dst, h-1) \quad (1)$$

    end if
  end function

Ideally GP would instantly compute a unique proxy X^c_{h-1}, if it exists, such that the proxy route through X^c_{h-1} is the best one possible. However, such an algorithm is unknown to us. Consequently, we examine a number of different widely-applicable strategies for proxy selection. Every version of GP that we explore is of the following form. Let (src, dst, h) be the inputs to GP. If $h = 0$ then GP outputs *null*; otherwise, let $h > 0$, so that *src* is in X^a_{h-1} and *dst* is in X^b_{h-1}, for some distinct a and b. GP computes a set of *candidate proxies* $\{X^{c_0}_{h-1}, X^{c_1}_{h-1}, \ldots, X^{c_{r-1}}_{h-1}\}$ (taken from the set of all potential copies of $X(h-1)$), and then finds a $c \in \{c_i : 0 \leqslant i \leqslant r-1\}$ for which the path in Expression (1) is of lowest cost (by constructing the paths explicitly); see Section 5.4 for some implementation details. If the set of candidate proxies is empty then GP returns *null*.

The key observation is that we must minimise the number of candidate proxies, in order to reduce the search space and execution time, but so that we do not overlook good proxies. Our goal is to identify and evaluate general techniques towards this end and not to catalogue all of the ways to tune GP. More complicated proxying techniques, such as proxying at more than one level, are therefore avoided; that is, we only evaluate PR where recursive calls to PR are effectively the same as DR. We describe three strategies for generating the candidate proxies below.

5.1 GP_E as an exhaustive search

A proxy copy of $X(h-1)$ can be obtained, naïvely, if GP is implemented as an exhaustive search; that is, we generate every possible copy of $X(h-1)$ as our candidate copies. Measuring the length of each resulting proxy route has an obvious associated cost, but GP_E does provide the optimal proxy route against which to test the two strategies given below.

5.2 GP_I as an intelligent search

We now propose a general method for reducing the proxy search space, based on the labels and uids of *src* and *dst*. In particular, we look at choosing proxies X^c_{h-1} whose relationship to X^a_{h-1} and X^b_{h-1} is such that at least one of the routes computed by the recursive calls to PR is 'short', by which we mean confined to a copy of $X(h-2)$ within $X(h-1)$. This is depicted in Fig. 2.

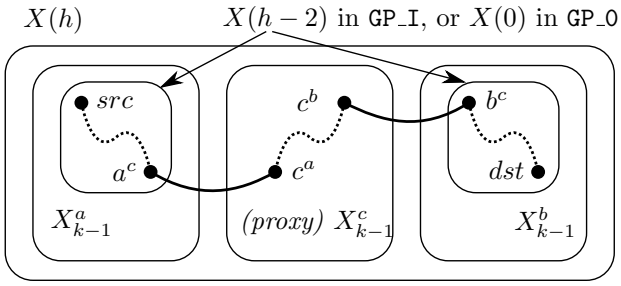


Figure 2: Strategy for GP_I and for GP_0. Solid arcs represent links, and dashed curves represent paths.

With reference to Section 3, every server-node x of $X(h)$ has a unique label of the form $x_h x_{h-1} \cdots x_0$ (and a unique uid $uid_h(x)$). Let *src* and *dst* be nodes in $X(h)$ with labels $a_h a_{h-1} a_{h-2} \cdots a_0$ and $b_h b_{h-1} b_{h-2} \cdots b_0$, respectively, where, w.l.o.g., $a_h < b_h$. So, within $X(h)$, *src* lies in $X^{a_h}_{h-1}$ and *dst* lies in $X^{b_h}_{h-1}$; furthermore, within $X^{a_h}_{h-1}$, *src* lies within $X^{a_{h-1}}_{h-2}$, and within $X^{b_h}_{h-1}$, *dst* lies within $X^{b_{h-1}}_{h-2}$.

Our algorithm GP_I does not necessarily build a set of candidate proxies if *src* and *dst* are already near to each other, where by 'near' we mean the following:

Condition 1 *src* is *near* to *dst* if the bridging link from $X_{h-1}^{a_h}$ to $X_{h-1}^{b_h}$ joins server-nodes that lie in $X_{h-2}^{a_{h-1}}$ and $X_{h-2}^{b_{h-1}}$.

Our intuitive reasoning is that copies of $X(h-2)$ cover sufficiently small neighbourhoods (especially when h is small; as we shall soon see, in our experiments in this paper $h \leqslant 3$) so that looking for proxies will generally not be particularly cost-effective. We apply Condition 1 only in the experiments described in Section 6.3, but see Section 5.4 for more discussion.

When GP_I does build a set of candidate proxies, it selects candidate proxies X_{h-1}^{c} for which at least one of Property 1 and Property 2 holds:

Property 1 the server-node in $X_{h-1}^{a_h}$ of the bridging link joining $X_{h-1}^{a_h}$ and X_{h-1}^{c} lies in $X_{h-2}^{a_{h-1}}$

Property 2 the server-node in $X_{h-1}^{b_h}$ of the bridging link joining $X_{h-1}^{b_h}$ and X_{h-1}^{c} lies in $X_{h-2}^{b_{h-1}}$.

Note that all server-nodes in $X_{h-2}^{a_{h-1}}$ have labels of the form $a_h a_{h-1} * * \cdots *$, where $*$ is arbitrary; and similarly all server-nodes in $X_{h-2}^{b_{h-1}}$ have labels of the form $b_h b_{h-1} * * \cdots *$. Of course, we wish to be able to efficiently compute the index c of each candidate proxy X_{h-1}^{c}. It is certainly the case that this can be done when $\mathcal{X}$ is DCell, Generalized DCell, and FiConn, as we now illustrate for DCell when $h = 3$.

From above, we have $a_3 < b_3$. Suppose that we are looking for some c such that $a_3 < b_3 < c$ and X_2^c is a candidate proxy. From the construction of DCell, a server-node in $X_1^{a_2}$ with label $a_3 a_2 x_1 x_0$ is adjacent to the server-node with uid a_3 in X_2^c where $c = uid_2(a_2 x_1 x_0) + 1$. Thus, for each pair $(x_0, x_1) \in \{0, 1, \ldots, n-1\} \times \{0, 1, \ldots, n\}$ for which $a_3, b_3 < uid_2(a_2 x_1 x_0) + 1$, we have a candidate proxy X_2^c, where $c = uid_2(a_2 x_1 x_0) + 1$. The cases where $a_3 < c < b_3$ and $c < a_3 < b_3$ are similar, but recall that the formula for computing c changes (see the connection rule for DCell). It is not difficult to see that: in the case that $a_2 \neq b_2$, there are at most $2n(n+1)$ candidate proxies (also, for FiConn there are at most $2n(n/2+1)$ candidate proxies); and in the case that $a_2 = b_2$ there are at most $n(n+1)$ candidate proxies (for FiConn there are at most $n(n/2+1)$ candidate proxies).

Notice that, in general, when we use Properties 1 and 2 to come up with our set of candidate proxies, we essentially restrict the portion of the path from *src* to *dst* that lies in $X_{h-1}^{a_h}$ or $X_{h-1}^{b_h}$ to be 'short' (so as to significantly limit the possibilities and under the intuition that doing so will still yield paths that are short overall). Note that we do not seek to consider candidate proxies X_{h-1}^{c} defined so as to restrict the portion of the path from *src* to *dst* that lies in X_{h-1}^{c} to be short, for the following reason: doing so would potentially lead to a very large number of candidate proxies, as a short path in X_{h-1}^{c} does not depend upon the parameters a_h and b_h; that is, our search space would not be appropriately narrowed.

5.3 GP_0 level-0 proxy search

When working with the method GP_I for $X(h)$, our notion of 'short' means being confined to some $X(h-2)$. When $h \geqslant 3$, we refine the method GP_I so as to redefine the notion of 'short' as being confined to some $X(0)$. This further limits the number of candidate proxies generated. We do this to study the trade-off between a reduction in computation time and a reduction in path quality. As we already highlighted, our experiments in this paper are all such that $h \leqslant 3$ and so GP_0 only differs from GP_I when $h = 3$.

5.4 Implementation notes

Below, we outline three aspects of PR that should be considered in any implementation.

First, when a proxy is chosen in the presence of faults or network congestion, hop-length may not be the only property of interest. In our version of PR, evaluated in Section 7, we compute the hop-lengths, connectivity, and (the number of flows routed through the) bottleneck link of each proxy-path, before choosing the path with the least congested bottleneck link from the set of shortest connected paths[2]. We might have instead chosen a shortest path from the set of paths with the least congested bottleneck link, or used some entirely different function of hop-length, connectivity, and congestion.

The second aspect pertains to the application of Condition 1. When there are no faults or highly congested links to avoid, path diversity is not required, and Condition 1 serves to reduce the search for proxies; the only experiment we report on that uses Condition 1 is the one described in Section 6.3 as regards

[2]This has no effect on hop-length experiments for PR in fault-free networks.

hop-length. Condition 1 is not applied at all in any of our other experiments. However, more sophisticated options exist as well. For example, one might first attempt to use DR when *src* and *dst* satisfy Condition 1, and explore proxy paths only if the initial attempt to route with DR fails.

The third aspect of PR is, more precisely, an optimisation, using table look-ups. The different choices alluded to above imply that a path-probing packet is used in order to ascertain the viability of alternative paths through the network; this technique is also used in FiConn [17]. We illustrate how table look-ups might be used to reduce the total travel of the probing packet using the example of $\text{DCell}_{3,n}$ and GP_I. Suppose that GP_I sends a probing packet from *src*, lying in $X_2^{a_3}$, towards *dst*, lying in $X_2^{b_3}$, through the proxy X_2^c. Let x and y be the server-nodes in X_2^c that are incident with the bridging links to and from X_2^c, respectively, on the proxy-path whose hop-length (and general viability) is being tested. A probing-packet moves from $X_2^{a_3}$ to x, containing administrative information such as where it has been or what its last hop was. If the (x, y)-path in X_2^c is congested, or contains a fault, the server-nodes visited by the probing packet have enough information to send a network status to x, regarding the (possibly temporary) viability of the path from x to y; *i.e.*, at no extra cost to the probing-packet process, the (x, y)-path status can be sent to and (possibly temporarily) stored in a table at x. This is by no means the only possible scheme, but just an example of how table look-ups and link-state, path-state, or even the state of whole $X(h)$s, can be used to reduce path-probe travel and increase efficiency.

6 Methodology

This section explains the methodology and experimental set-up we use to carry out our exhaustive analysis and comparison of PR with previously-known routing algorithms for (β-)DCell and FiConn, which are henceforth referred to collectively as $\mathcal{G}$-Cell. We examine both raw performance metrics, including hop-length distributions, throughput, and fault-tolerance, as well as application-oriented performance metrics, including zero-load latency and workload completion time. Our evaluations consider up to five different routing algorithms for various $\mathcal{G}$-Cells as well as a wide range of representative workloads. We test PR on the full range of level 2 and 3 $\mathcal{G}$-Cells from around $15,000$ to 3.5 million server-nodes. More specifically, we study $\text{DCell}_{k,n}$ and $\beta\text{-DCell}_{k,n}$ with $k = 2$ and $11 \leqslant n \leqslant 43$, and with $k = 3$ and $3 \leqslant n \leqslant 6$; and $\text{FiConn}_{k,n}$ with $k = 2$ and $22 \leqslant n \leqslant 86$, and with $k = 3$ and $8 \leqslant n \leqslant 16$, for even n.

Some representatives of these networks, and the ones we are focusing on in most of the experiments, are shown in Table 1. The table also shows the number of potential proxies at each level and the number of server-nodes and links.

6.1 Software tool: INRFlow

The main body of our experiments is conducted with our software tool Interconnection Networks Research Flow Evaluation Framework (INRFlow) [8]. INRFlow is an open-source *flow-level* network simulator, implemented in C, that enables the evaluation of the performance of a given network under various conditions. It supports a wide variety of networks, routing algorithms, and workloads and, indeed, includes an implementation of $\mathcal{G}$-Cell. INRFlow operates as follows: for each flow (*i.e.*, a source-destination pair of server-nodes and possibly an amount of data), a route is computed with a custom routing algorithm (PR or DR in our case) and link-utilisation is recorded. When all flows have been routed, the simulation is complete and INRFlow reports a large number of statistics.

There are both *dynamic* and *static* execution modes. In dynamic mode, the amount of data to be communicated is provided with each source-destination pair, as well as the bandwidth capacity of the links. INRFlow's dynamic engine performs a realistic simulation of the flows traversing the network, while preserving message causality, *i.e.*, respecting the temporal relationships between sends and receives. The output provides information about the execution of the workload, such as flow latency or throughput, and the completion time of the whole workload.

In static mode, the bandwidth capacities of links are ignored, all flows are routed, and the network usage is analysed. These experiments serve to analyse properties where the interaction among flows is less important, such as hop-length or connectivity in faulty networks. They are inherently much faster than dynamic experiments, so we can use them to study much larger networks than we can with dynamic mode.

DCN	N	N/n	$\|E\|$	d	g_1	g_2	g_3
$F_{2,36}$*	117648	3268	161766	7	19	172	
$F_{2,48}$	361200	7525	496650	7	25	301	
$F_{3,10}$*	116160	11616	166980	15	6	16	121
$F_{3,16}$	3553776	222111	5108553	15	9	37	667
$D_{2,18}$*	117306	6517	234612	7	19	343	
$D_{2,43}$	3581556	83292	7163112	7	44	1893	
$D_{3,3}$	24492	8164	61230	15	4	13	157
$D_{3,4}$*	176820	44205	442050	15	5	21	421
$D_{3,6}$	3263442	543907	8158605	15	7	43	1807

Table 1: Properties of the DCNs in our experiments. In tables and plots we use F to abbreviate FiConn, and D to abbreviate (β-)DCell. DCNs used in the experiments of Section 6.7 are marked with (*). N: server-nodes, N/n: switch-nodes, $|E|$: links, d: the dimensional route upper bound, g_i: potential proxies at level i.

6.2 Workloads

We now describe the traffic patterns used in our evaluation. FiConn and DCell were designed to meet the needs of exascale tenanted cloud datacenters which can serve in the utility- or cloud-computing model described, for example, by Armbrust *et al.* [3]. Therefore we focus on traffic patterns and workloads arising from many simultaneous data-intensive applications which would typically employ a MapReduce-paradigm; see, for example, [5,19,23].

Multi-application environments are simulated through the *many all-to-all* traffic pattern, where N server-nodes of the network are partitioned into $\lfloor N/g \rfloor$ groups of g server-nodes, and each group performs an all-to-all communication. The groups are generated uniformly at random, without knowledge about the network topology, which reflects major operators' policies of distributing storage across "clustered subsets" of the network in order to reduce the effects of correlated failures [20] (the definition of a clustered subset varies from one topology to another, of course, but we are simply emphasising that applications are often uncorrelated, or even anti-correlated, with the network's natural division into low-diameter substructures).

To model the worst-case situation where all of the servers are workers sending their results back to a single master server and, consequently, saturating the network around a single *hot-spot*, we consider the *one-to-all* traffic pattern.

Finally, to cover unstructured applications (*e.g.*, graph analytics) we consider *uniform random* traffic, where 100 million source-destination pairs of server-nodes are chosen uniformly at random. This pattern represents a scenario where there is a large volume of traffic with no spatial locality, resulting in many localised bottlenecks appearing due to the high levels of contention for the use of resources.

6.3 Mean hop-length experiments

In Section 2 we explain that a hop is either a server-node-to-server-node link or a server-node-to-server-node path that passes through a switch-node. Thus, we study *hop-length*, *i.e.*, the number of hops in each path, as is common practice in the community.

Hop-length impacts four of the most important aspects of the DCN. Lowering mean hop-length reduces the total amount of traffic in the network, thereby lowering the overall usage of network resources. Consequently, energy usage — one of the highest costs associated with DCNs — is also reduced. Additionally, the number of hops travelled by an individual packet (or frame) has a direct impact on its latency (see Section 6.6).

In our hop-length experiments, we compare: `DR`; shortest paths (as regards hop-length), computed by a breadth first search (`BFS`); and `PR` with `GP_E`, `GP_I`, and `GP_O`. In all instances of `GP` we apply Condition 1. Each routing algorithm (for a given DCN) is tested with the same $10,000$ input pairs, (src, dst), chosen uniformly at random.

We also study the number of proxies explored by `GP_I` and `GP_O`, denoted $\bar{p}$ `GP_I` and $\bar{p}$ `GP_O`, respectively, and the number of proxy routes found to be no longer than the dimensional route in each of `GP_E`, `GP_I`, and `GP_O`, denoted $\bar{r}$ `GP_E`, $\bar{r}$ `GP_I`, and $\bar{r}$ `GP_O`, respectively.

Our hop-length results (see [9]) were obtained be-

fore the development of INRFlow was completed and so used a purpose-built tool written in Python using the Sage computer algebra system. However, we have verified them with INRFlow.

6.4 Fault-tolerance and load-balance experiments

Graceful performance degradation in the presence of network faults is important in DCNs, as faults become commonplace with large numbers of components. In our experiments on fault-tolerance we focus on the connectivity achieved by the different routing algorithms when 10% of the links, selected uniformly at random, fail. We also investigate what effect the appearance of failures has on the lengths of the paths between communicating server-nodes.

6.5 Network throughput experiments

Big Data applications typically produce large numbers of data-heavy flows and require a tremendous amount of throughput as data is moved in the datacenter. Aggregate bottleneck throughput (ABT), introduced in [11], is a metric used to evaluate the throughput performance of a topology under worst-case, heavy traffic conditions. We define the *Restricted Aggregate Throughput (RAT)* as $\Theta_R = Fb/B$, for a given traffic pattern with F flows, where B is the number of flows using the network's bottleneck link, and b is the bandwidth of each link. The RAT indicates how well a DCN performs for applications where the tasks are tightly coupled and synchronise often, and the slowest flow restricts the pace at which the application can advance. The RAT is similar to the ABT considered in [11], which is defined only for the all-to-all traffic pattern (in which case the ABT and the RAT are identical). We define the *Unrestricted Aggregate Throughput (UAT)*, $\Theta_U = Fb/\bar{B}$, where $\bar{B}$ is the average number of flows in each link of the network. The UAT indicates how well a DCN performs for self-throttling applications in which the synchronisation component is minimal and the data associated with the flow adapts to the available bandwidth. In this case, the tasks are able to make the most of the available throughput and so Θ_U is derived from the mean congestion instead of from the bottleneck. The UAT is akin to the throughput metric proposed in [24], called LFTI. In our experiments we focus on both RAT and UAT.

6.6 Latency experiments

While datacenters tend to be used as stream-processing systems, and so are typically more susceptible to throughput variations, there are also many datacenter applications which are more sensitive to latency, *e.g.*, real-time operations or applications with tight user interactions such as real-time game platforms, on-line sales platforms, and search engines.

For this reason, we also look at the end-to-end latency of each algorithm. We base our analysis on the latencies imposed by the different steps of the communication: protocol stack, propagation latency, data transmission latency, and routing at the servers. All of the *transmission latencies*, *i.e.*, protocol stack, propagation, and data, are measured using the standard UNIX ping utility, whereas the *routing latency* is measured within INRFlow (we say more about the different latencies in Section 7.4. All the transmission latency measurements are carried out independently under low load conditions in the same server, a 32-core AMD Opteron 6220 with 256GB of RAM and running Ubuntu 14.04.1 SMP OS. The server and its neighbour are located in the same rack and are connected with short (< 1 mtr.) electrical wires to a 24-port 1Gbit Ethernet switch. This platform is used because it is representative of COTS hardware. In this configuration we actually measure lower bounds on transmission latencies, since we do not consider other instrumentation needed for a server-centric architecture over and above short wires and protocol stack latency. We measure the routing latency with INRFlow in level 3 topologies with around 100K server-nodes (namely $\text{FiConn}_{3,10}$ and $\text{DCell}_{3,4}$).

Note that routing time measured with INRFlow provides a conservative estimate of routing latency that benefits DR and penalises PR. In a real-world implementation of PR, where latency is truly critical, a number of optimisations could be applied that would reduce the overheads of PR (see Section 5.4), relative to those of DR. Thus, since our measured routing latency is an upper bound, and our measured transmission latency is a lower bound, the real proportion of routing latency to transmission latency would be smaller than it is in our measurements. Consequently, hop-length reduction will have a greater impact on the overall latency.

6.7 Completion time experiments

Ensuring low completion-time of workloads under a variety of conditions is arguably the ultimate objec-

tive in the design of routing-algorithms, and therefore we perform experiments to evaluate the performance of PR in this regard using INRFlow's dynamic engine. We measure the time required to process all the traffic that composes a workload, that is, the time required by all flows from all the sources to be delivered to their destinations. By this we ascertain whether the applications are able to make the most out of the improvements in terms of path length and throughput offered by PR and consequently run to completion in a shorter time. The size of the flows and the speed of the network links has an impact on the results; thus, we have used flows of 1GB and links of 1Gbps as good representatives of the kind of hardware and workloads that we could expect in an SCDCN.

We compare DR against PR with GP_E, GP_I, and GP_O, for those DCNs with $k = 2$ and $k = 3$, marked with a (*) shown in Table 1, taking the average of 10 trials.

6.8 Sampling error

Sampling error arises both by using sample populations of randomly selected flows, and by choosing sets of failed links. Small sampling errors are broadly unimportant in our analysis, but nevertheless we try to minimise sampling error. Firstly, our flow-sample size of 100 million is very large, relative to the sample standard deviation of any given statistic we present (the experiments of Section 6.3, originally appearing in [9], were verified using INRFlow with a 100 million flow sample size). Secondly, whereas only one faulty network is chosen for each 100 million flow simulation, the uniformity of the DCNs we are studying provides that there is very little overall difference between the effects of one set of faulty links and another, *i.e.*, the standard deviation is negligible, so the number of trials is not important. Indeed this is reflected in the low amount of statistical noise seen in our data (including data that is discussed but is not plotted in our paper for the sake of brevity). Whilst quantifying this error is outside the scope of our paper, it is evident from the low amount of noise in our plots that the true error is negligible in the context of the conclusions we are making, and thus, we omit error bars from the plots for simplicity.

7 Experimental evaluation

Henceforth, we often refer to PRwith, for example, GP_I simply as GP_I.

7.1 Hop-length evaluation

The plots in Fig. 3 show that for many $\mathcal{G}$-Cell topologies, significant savings in hop-length can be made over dimensional routes by using proxy routes, depending on the connection rule, network size, and the parameters k and n. It is immediate that GP_I and GP_O retain some good proxies, in relation to GP_E, which tries all of them. Furthermore, the savings obtained with GP_E consistently approach, and in some cases equal, those of BFS. Fig. 4 tells us how much searching each of the methods GP_I and GP_O must do, and how much path diversity they create, on average, in fault-free networks.

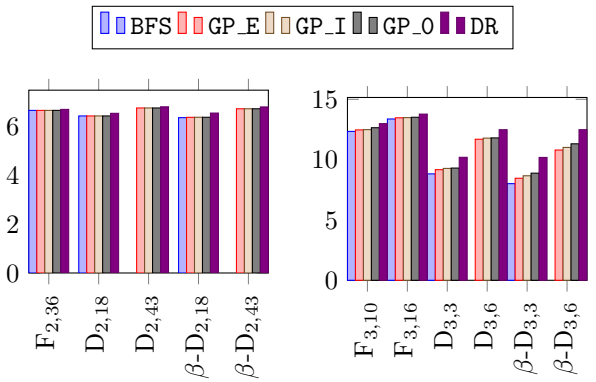


Figure 3: Mean hop-length. BFS experiments for large networks with millions of server-nodes were prohibitive both in terms of computation time and memory and thus are not shown in the plot.

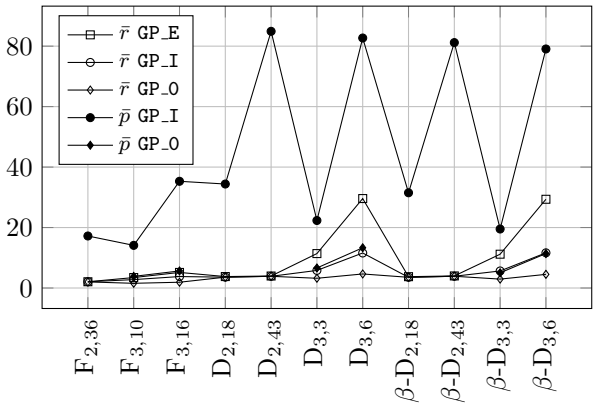


Figure 4: Mean number of candidate proxies, $\bar{p}$, and mean number of routes no longer than DR(src, dst), $\bar{r}$.

Note that the means plotted in Figs. 3 and 4 hide

the success rate of PR in finding a good proxy path; as a typical example, PR(src, dst) is shorter than DR(src, dst) for approximately 30% of input pairs when using GP_I in DCell$_{3,6}$.

We highlight (and explain, where possible) some of the trends observable in the plot of Fig. 3. In general, proxy routes are more effective in β-DCell$_{k,*}$ than in DCell$_{k,*}$ and FiConn$_{k,*}$ of comparable size, with fixed k; however, even FiConn$_{k,*}$ still sees up to a 6–7% improvement.

The apparent weakness of PR in FiConn is partly explained by the fact that for given k and n, there are fewer proxy FiConn$_{m-1,n}$s to consider at level m. For example, GP_0 considers only 3.9 proxies, on average, for FiConn$_{3,10}$, while it considers around 12 proxies, on average, for DCell$_{3,6}$ and β-DCell$_{3,6}$. On the other hand, there are equal numbers of potential proxies in β-DCell$_{k,n}$ and DCell$_{k,n}$ in general, yet GP_I and GP_0 invariably consider more proxy candidates for DCell$_{k,n}$, only to produce proxy paths that perform better in β-DCell$_{k,n}$. We must conclude that the connection rule and topology profoundly impacts the performance of our proxy routing algorithms. This is somewhat unsurprising, however, since the connection rule and topology also affect the hop-lengths of the shortest paths; for example, the mean distance in β-DCell$_{3,3}$ is far shorter than in DCell$_{3,3}$ (as is also observed in [16]).

Proxy paths in larger networks (when increasing n) are longer than those in smaller networks, for each DCN with fixed k; for example, compare DCell$_{3,3}$ and DCell$_{3,6}$, and also FiConn$_{3,10}$ and FiConn$_{3,16}$. Intuitively, the reason for this is that as n increases, the proportion of good proxies reachable from within a level-0 (or level-1 for GP_I) sub-DCN decreases.

A related trend appears to be that for each family of DCNs, proxy-path savings increase with k, in every version of GP. The main reason for this is that the performance of BFS, relative to DR, also increases with k, thus providing a greater margin for improvement by using PR.

The differences in performance between GP_I and GP_0, and between GP_E and GP_I, grow with[3] k. The double exponential growth of $\mathcal{G}$-Cell results in the double exponential growth of g_i, for increasing i, while the search spaces for GP_E, GP_I, and GP_0 are proportional to g_{k-1}, g_{k-2}, and g_0, respectively (see Table 1). The algorithms with more candidate proxies to choose from, understandably, tend to perform better. Note, however, that for $\mathcal{G}$-Cell$_{2,*}$, the performance of GP_E is almost identical to the performance of GP_I: whereas DCell$_{2,43}$ has $g_1 = 44$ and $g_2 = 1893$, our results show that optimal proxies are nevertheless considered by GP_I (and, hence, GP_0).

Although PR is effective in computing shorter paths and comes fairly close to BFS we can confirm that the shortest paths for these topologies are not always general routes of order at most 3, which motivates future research on general routes of order 4 and higher.

[3]Remember that for $k = 2$, GP_I and GP_0 are identical.

7.2 Fault-tolerance and load-balance evaluation

We evaluate the resiliency of PR with GP_I in faulty networks as well as its ability to balance traffic-loads (see Section 5.4).

Figs. 5 and 6 show that GP_I performs far better than DR in faulty networks. It is unsurprising that DR performs poorly here since DR does not adapt to faults; however, GP_I's connectivity of 80–90% in level-2 $\mathcal{G}$-Cells, as well as level-3 DCells and β-DCells is impressive. The proxy paths offer a significant amount of link-diversity for circumventing link-faults. The results of our fault-tolerance experiments for many all-to-all and one-to-all traffic patterns are consistent with these results and are therefore not shown.

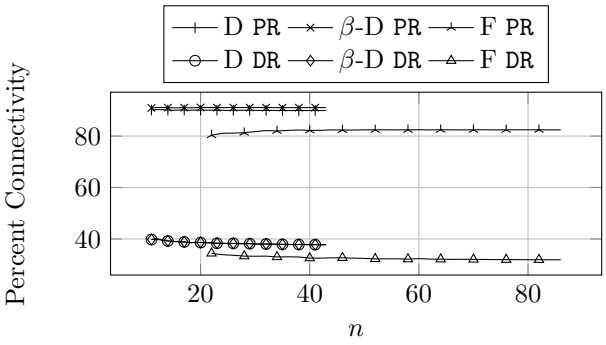


Figure 5: Connectivity of GP_I and DR for random traffic with 100 million flows and 10% faulty links when $k = 2$. Note that some of the marks are omitted from the plot for clarity.

The (link) fault-tolerance of GP_I is comparable to that of DFR, as evidenced by the data reported in [12], but GP_I outperforms DFR as regards hop-length. In particular, for DCell$_{3,4}$, DFR achieves around 95% connectivity when 10% of the links have failed, and GP_I provides around 85%. The high connectivity of DFR, however, comes at the cost of path-length: the respective mean hop-lengths of paths routed by

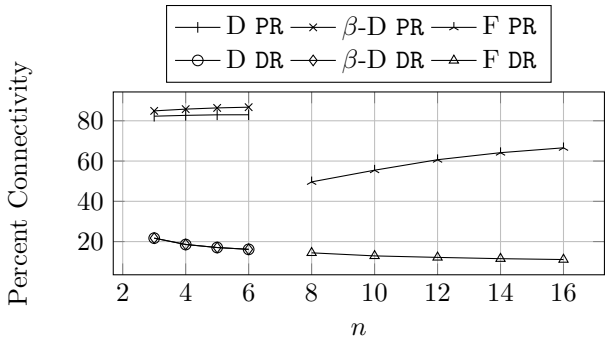


Figure 6: Connectivity of GP_I and DR for random traffic with 100 million flows and 10% faulty links when $k = 3$.

DFR[4] and GP_I are 14 and 10.7. In the absence of faults DR routes paths of average hop-length 10.2; that is, DFR incurs a path-length penalty of over 35%, whereas GP_I incurs 5%. For β-DCell$_{3,4}$ the path-length penalty for GP_I is 0%; recall that no fault-tolerant algorithm has been developed for β-DCell as of this writing.

Figs. 7 to 9 show hop-length distribution histograms for three level-3 $\mathcal{G}$-Cells that we are considering. Other level-3 networks have similar hop-length distributions. As expected from the results of Section 7.1, the histograms show that without faults, GP_I generates shorter routes than DR. Most significant, however, is that the paths found by GP_I in a 10% faulty network, are not much worse than those found by DR in a fault-free network, with only a few of the GP_I-routes being much longer than the maximum length DR-route, *i.e.*, 15 hops. In Fig. 9 there is a greater number of longer GP_I-routes, where most of them are no longer than 19 hops; however, nearly 35% of the routes found by DR in FiConn$_{3,10}$ are already 15 hops, so this is unsurprising. This is a very promising result, since the existing fault- and load-tolerant routing algorithms TAR and DFR produce significantly longer paths than DR when faced with faults and congestion. We have already described above the relevant previous results for DFR and in the results for FiConn of [17], TAR computed paths that are 15-30% longer, on average, than those computed by TOR (see also Theorem 7 of [17]).

The parity artefacts for FiConn in Fig. 9, where there tend to be more odd-length than even-length paths, is (partly) caused by the dual-port property of FiConn. The hops on any path in FiConn$_{k,n}$ necessarily alternate between level-0 hops and level-d hops, for some $0 < d \leqslant k$. Thus, the parity of the path-length depends on whether both, neither, or exactly one of its terminal hops are level-0 hops, and the distribution of the parities of path lengths relates to the distribution of paths with these same properties. The data show that both GP_I and DR are less likely to route paths with exactly one level-0 terminal hop.

Figs. 10 and 11 show that GP_I indeed reduces the overall network usage in (β-)DCell, as expected from the results on hop-length, and that it also reduces the number of flows in the most congested link(s). A less congested bottleneck can also increase throughput (see Section 7.3).

We offer an intuitive explanation for the results plotted in Fig. 12, that show that GP_I does not reduce the bottleneck congestion in FiConn. The structure of FiConn$_{k,n}$ is such that each server-node is connected to a switch-node by a level-0 link, and, possibly, to a server-node by a level-d link, for some $d > 0$; we call this a *level-d server-node*. For each $0 < d < k$, there are twice as many level-d server-nodes as there are level-$(d+1)$ server-nodes. For a (src, dst)-pair of server-nodes in distinct FiConn$_{k-1,n}$s, a proxy-route uses twice as many level-k links as a dimensional route, and thereby shifts the burden of network traffic onto a smaller set of links. Consider, for example, the two paths highlighted in FiConn$_{2,4}$, shown in Fig. 1: the dimensional route uses 4 level-0 hops, 2 level-1 hops, and 1 level-2 hop, whilst the proxy-route uses 2 level-0 hops, 1 level-1 hop, and 2 level-2 hops. Note that the distribution of flows in level-0 links is also changed, since each level-0 link is associated with the level-d link that connects to the same server-node (if the server-node is of degree 2), and we can observe the same congestion in these links as we do in the higher level links. The congestion in high level links (and associated level-0 links) results in the bottlenecks seen in Fig. 12.

7.3 Network throughput evaluation

This section explores how the different routing algorithms affect network throughput. Figs. 13a and 13b show the gains made for the UAT, Θ_U, and RAT, Θ_R, respectively.

If we focus on the UAT then we can see that all the networks benefit from every version of GP. The improvement in Θ_U for FiConn$_{k,n}$, with $k \in \{2, 3\}$, and (β-)DCell$_{2,n}$ is below 5%; however, it is as high as 30% for (β-)DCell$_{3,n}$. In the latter case, considerable improvement is obtained by GP_0, with GP_I yielding

[4]Interpolated conservatively from Table 2 of [12].

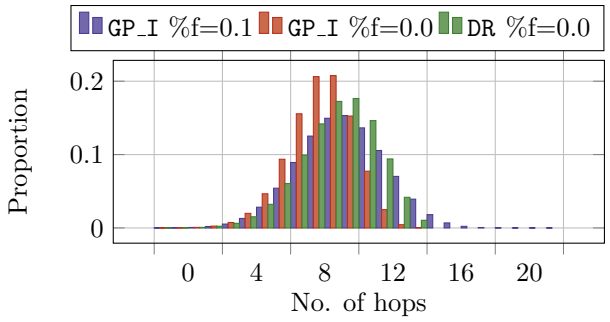


Figure 7: Hop-length histograms for $\text{DCell}_{3,3}$ under random traffic with 100 million flows, comparing GP_I with DR. "%f" stands for percent failed links; normalised.

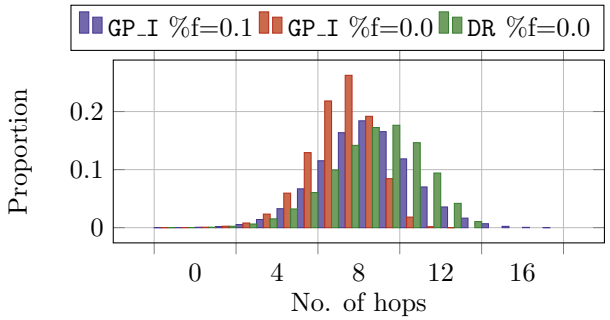


Figure 8: Hop-length histograms for β-$\text{DCell}_{3,3}$ under random traffic with 100 million flows, comparing GP_I with DR. "%f" stands for percent failed links; normalised.

about 5% more, and, finally, with GP_E yielding an additional 5% more.

PR has no effect on the RAT of $\text{FiConn}_{2,36}$, and it has, in fact, a negative effect on $\text{FiConn}_{3,10}$; see Section 7.2, where we explain the origin of the bottlenecks that bring a reduction in RAT. It is usually the case that for (β-)DCell, the improvement in RAT is much greater than the improvement in UAT, and can be up to an astonishing 55%. The improvement in load-balance is discussed in Section 7.2.

The differences in RAT between GP_E, GP_I, and GP_0 tend to be larger than the differences in UAT. The reason for this is that the larger the exploration space in terms of proxies, the more homogeneous the use of network resources. This yields a more balanced use of the network that, together with the availability of shorter paths, is translated into overall better utilization of network bandwidth and higher throughput.

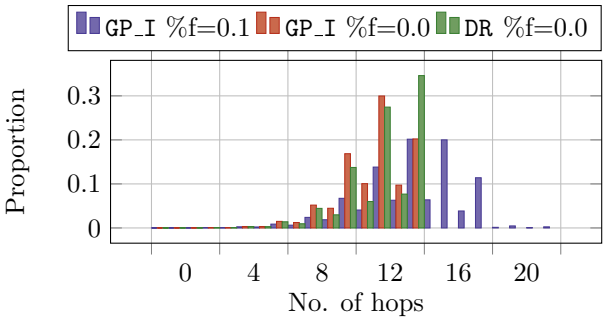


Figure 9: Hop-length histograms for $\text{FiConn}_{3,10}$ under random traffic with 100 million flows, comparing GP_I with DR. "%f" stands for percent failed links; normalised.

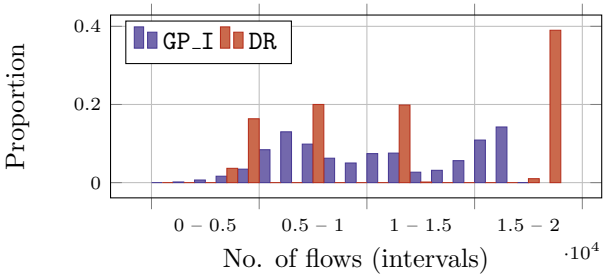


Figure 10: Flow histogram for $\text{DCell}_{3,3}$ under random traffic with 100 million flows, comparing GP_I with DR in a fault-free network. Zero-flows omitted.

7.4 Latency evaluation

This section studies the effect of the different algorithms on communication latency, as per the experiments outlined in Section 6.6. The contributors to the overall latency are measured as follows.

- The *stack latency*, L_s, is derived by measuring the round-trip time of both an *empty* frame (28 bytes for the headers) and a *full* frame (1,500 bytes, including the headers) sent to `localhost`. In both cases L_s is 10μs.

- To derive the *propagation latency*, L_p, we measure the round trip of an empty frame sent to another server connected to the same Gigabit Ethernet switch (64μs). Dividing this figure by two and subtracting L_s, we get an estimate of 22μs.

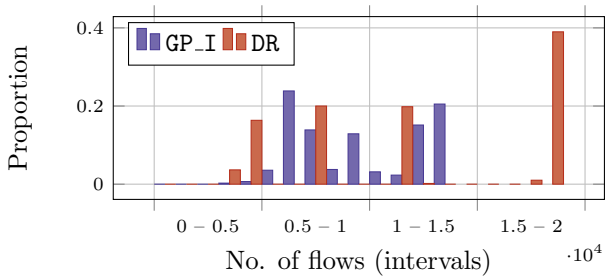


Figure 11: Flow histogram for β-DCell$_{3,3}$ under random traffic with 100 million flows, comparing GP_I with DR in a fault-free network. Zero-flows omitted.

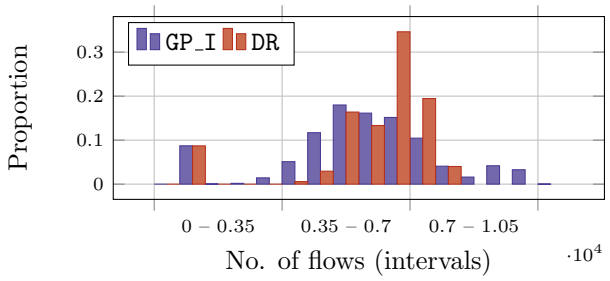
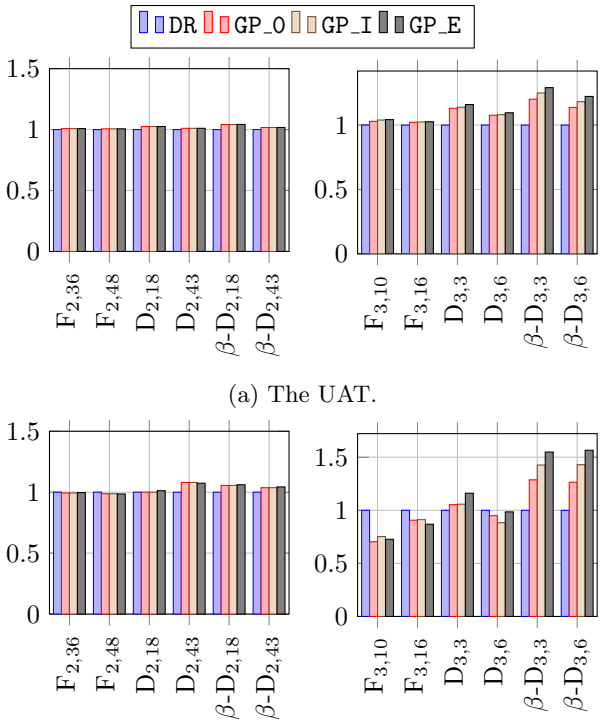


Figure 12: Flow histogram for FiConn$_{3,10}$ under random traffic with 100 million flows, comparing GP_I with DR in a fault-free network. Zero-flows omitted.



(a) The UAT.

(b) The RAT.

Figure 13: Performance comparison in terms of throughput. Values normalized to DR.

- Similarly, we derive the *data transfer latency*, L_d, by measuring the round trip time of a full-frame sent to the same neighbour server (140μs). Similarly dividing by two and subtracting L_p and L_s, we get 38μs per full frame.

- We derive the *routing latency* by measuring the average running time of the algorithms per hop, L_r, for each of the topologies (averaged over half a million randomly generated flows). The measurement scheme implemented in INRFlow has a resolution of nanoseconds and the measured values have (at least) 4 significant digits; hence, our measurements, when expressed in integer milliseconds, are accurately portrayed.

Adding these measurements we can compute the per-hop latency $L_H = L_s + L_p + L_d + L_r$. Multiplying L_H by the average hop-length obtained by each algorithm we get an estimation of the zero-load routing latency for the different routing algorithms and topologies as per Table 2. The zero-load routing latency is not highly realistic; however it provides a general indication of how much the overheads of proxy-searching affects latency when the savings in hop-length are taken into account. We conducted the same latency experiments for smaller topologies, and the results are consistent with the ones shown in Table 2.

Table 2: Average zero-load latencies in milliseconds.

	DR	GP_0	GP_I	GP_E
FiConn$_{3,10}$	0.913	0.894	0.908	1.122
DCell$_{3,4}$	0.795	0.752	0.796	1.400
β-DCell$_{3,4}$	0.794	0.714	0.742	1.361

Recalling the discussion in Section 6.6, our methodology handicaps PR, since we have not implemented every possible optimisation for PR in INRFlow; Table 2 lists conservative estimates on the latencies for GP_0, GP_I, and GP_E. On the other hand, DR has a very simple implementation with little room for optimisation, so Table 2 lists more realistic estimates for DR. Keeping this in mind, we nevertheless interpret the actual figures, as follows.

The results in Table 2 show negligible to medium

improvements in latency for each topology when using GP_0 and GP_I, while a considerable penalty is incurred by GP_E. GP_0 shows a latency reduction of between 2% and 10%, whereas GP_I provides negligible differences for FiConn and DCell and a 7% improvement for β-DCell. Using GP_E slows down packet transmission significantly (up to 75% in the worst case) because of the high number of proxy candidates it needs to explore. We conclude that while improvements obtained with GP_E are offset by the extra latency introduced by the proxy search, the guided selection of proxies introduced in this paper (GP_I and GP_0) suits latency-sensitive applications.

7.5 Completion time evaluation

This section discusses the completion time experiments described in Section 6.7. Completion times (in seconds) for different workloads and topologies are shown in Fig. 14.

The results for random uniform and many all-to-all traffic patterns are poor in FiConn, in spite of the fact that the routes computed by PR are shorter than those computed by DR. This may seem surprising, but it is explained by our previous observations on load-balance, in Section 7.2. The gains made by PR for the (lighter) one-to-all traffic pattern are also consistent with our earlier assessment of congestion in FiConn. The bottleneck congestion occurs in links near the source node of the one-to-all communication, regardless of which routing algorithm is used, and the low amount of bandwidth available on each of these links dominates the completion time. Areas of the network further away from the source server-node are not highly loaded, so PR makes marginal gains over DR by routing on shorter paths.

With DCell and β-DCell, the use of PR always improves upon DR. The results depicted in Fig. 14 for DCell and β-DCell confirm that proxy-based routing algorithms achieve lower completion times than DR for level 2 and 3 DCNs. As expected, GP_E (due to the balanced use of the network and the use of shortest paths) results in the best routing algorithm for both level 2 and 3 DCNs.

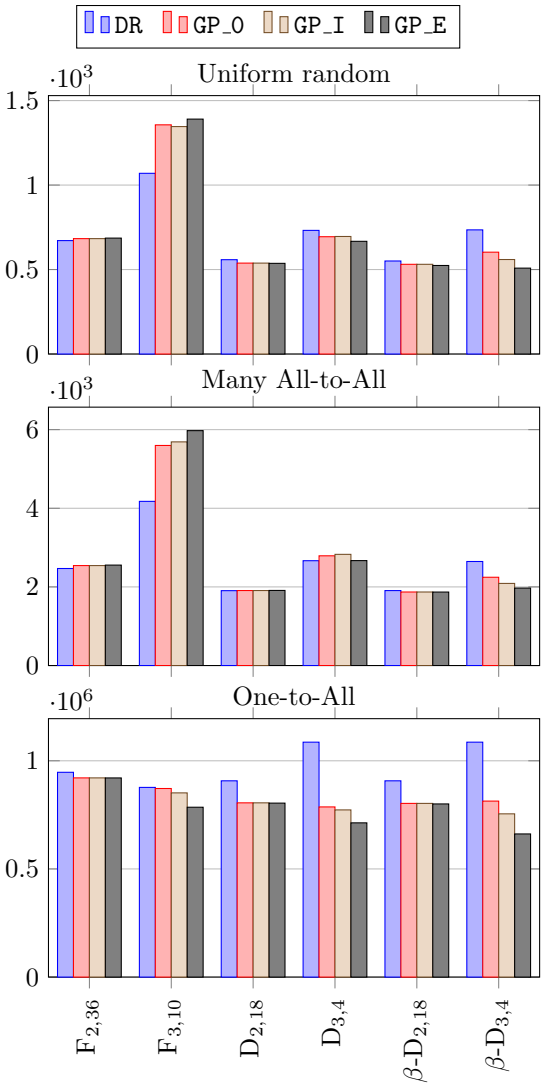


Figure 14: Completion time in second employed to process three traffic patterns comparing DR, GP_0, GP_I and GP_E.

8 Conclusions and future research

We have shown that the DCNs (Generalized) DCell and FiConn are completely connected recursively-defined networks and, as such, we have characterised all possible routes (with no repeated nodes) in these networks and then proposed the family of routing algorithms PR to efficiently compute the class of routes called proxy routes. We detailed three alternative subroutines of PR, namely GP_E, GP_I, and GP_0, that search for and select an optimal proxy to route through. We then performed a comprehensive analytical and empirical comparison between PR and other existing routing algorithms for (Generalized) DCell and FiConn, on the basis of hop-length, fault-tolerance, load-balance, network-throughput, end-to-end latency, and workload completion time.

Our evaluation shows that for most of the topologies we have considered, PR exhibits strong performance with respect to all of the above metrics, outcompeting existing routing algorithms in almost every aspect, even when using GP_I or GP_0, which only search a small set of candidate proxies. The performance gains are achieved by efficiently computing shorter routes, and using the path-diversity naturally introduced by the proxy-search to avoid link failures and balance network traffic.

The gains in hop-length, load-balance, and throughput were comprehensively demonstrated in dynamic, flow-level simulations using the simulator INRFlow. We have shown that PR completes a number of realistic benchmark DCN workloads up to 40% faster in β-DCell, and up to 35% faster in DCell, than the existing dimensional routing algorithm. Although the dual-port property severely inhibits PR in FiConn by increasing network congestion when proxies are used for routing, light workloads nevertheless benefit from the shorter paths computed by PR.

In future research we will perform a deeper analysis of the DCNs in question, with two major goals. The first one, motivated by the fact that GP_I sometimes discards the optimal proxy candidate, calls for a closer theoretical inspection of the topologies. We want to both find the optimal proxy candidates, and reduce the size of the search space.

Furthermore, whereas this paper is focused on dimensional and proxy routing, there may be cases where no shortest path between two server-nodes is a dimensional route or a proxy route. Note that exhibiting a shortest path that is neither a proxy route nor a dimensional route does not preclude the existence of another shortest path from being a proxy route or dimensional route. A deeper mathematical analysis of the DCNs in question may shed light on (1) whether or not higher-order routing algorithms are needed, and (2) how to compute optimal higher order routes efficiently.

Finally, our results have demonstrated that with Generalized DCell, it does matter how one chooses the interconnection rule. The two connection rules studied here are by no means the only ones available, and if one has invested in a Generalized DCell DCN then it would be useful to better understand its performance under a range of different interconnection rules. Similarly, it would be interesting to study FiConn under different interconnection rules too.

Acknowledgements

This work has been funded by the Engineering and Physical Sciences Research Council (EPSRC) through grants EP/K015680/1 and EP/K015699/1. Dr. Navaridas is supported by the European Union Horizon 2020 research and innovation programme under grant agreement No 671553.

References

[1] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. *ACM SIGCOMM Computer Communication Review*, 40(4):51–62, Aug. 2010.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM Conference on Data Communication*, pages 63–74, New York, NY, USA, Aug. 2008. ACM.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, Apr. 2010.

[4] G.-H. Chen, S.-C. Hwang, M.-Y. Su, and D.-R. Duh. A general broadcasting scheme for recursive networks with complete connection. In *Proc. of International Conference on Parallel and Distributed Systems*, pages 248–255, Dec. 1998.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[6] R. Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate Texts in Mathematics*. Springer, 2012.

[7] D.-R. Duh and G.-H. Chen. Topological properties of WK-recursive networks. *Journal of Parallel and Distributed Computing*, 23(3):468–474, Dec. 1994.

[8] A. Erickson, A. Kiasari, J. Pascual Saiz, J. Navaridas, and I. A. Stewart. Interconnection Networks Research Flow Evaluation Framework (INRFlow), 2016. [Software] https://bitbucket.org/alejandroerickson/inrflow.

[9] A. Erickson, A. E. Kiasari, J. Navaridas, and I. A. Stewart. Routing algorithms for recursively-defined data centre networks. In *Proc. of Trustcom/BigDataSE/ISPA, IEEE*, volume 3, pages 84–91. IEEE, Aug. 2015.

[10] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 41(4), Aug. 2010.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, Aug. 2009.

[12] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. *ACM SIGCOMM Computer Communication Review*, 38(4):75–86, Aug. 2008.

[13] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, and G. Chen. Expandable and cost-effective network structures for data centers using dual-port servers. *IEEE Transactions on Computers*, 62(7):1303–1317, Jul. 2013.

[14] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. *SIGARCH Computer Architecture News*, 35(2):126–137, June 2007.

[15] M. Kliegl, J. Lee, J. Li, X. Zhang, C. Guo, and D. Rincón. Generalized DCell structure for load-balanced data center networks. In *INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–5, Mar. 2010.

[16] M. Kliegl, J. Lee, J. Li, X. Zhang, D. Rincon, and C. Guo. The generalized DCell network structures and their graph properties. Microsoft Research, October 2009.

[17] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, S. Lu, and J. Wu. Scalable and cost-effective interconnection of data-center servers using dual server ports. *IEEE/ACM Transactions on Networking*, 19(1):102–114, Feb. 2011.

[18] Y. Liao, J. Yin, D. Yin, and L. Gao. DPillar: Dual-port server interconnection network for large scale data centers. *Computer Networks*, 56(8):2132–2147, May 2012.

[19] Y. Liu, J. K. Muppala, M. Veeraraghavan, D. Lin, and M. Hamdi. *Data Center Networks: Topologies, Architectures and Fault-Tolerance Characteristics*. Springer, 2013.

[20] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Proc. of ACM SIGCOMM Conference on Data Communication*, New York, NY, USA, 2015. ACM.

[21] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of 9th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2012. USENIX Association.

[22] G. D. Vecchia and C. Sanges. A recursively scalable network VLSI implementation. *Future Generation Computer Systems*, 4(3):235–243, Oct. 1988.

[23] T. White. *Hadoop: the Definitive Guide*. O'Reilly Media, Inc., 2009.

[24] X. Yuan, S. Mahapatra, M. Lang, and S. Pakin. LFTI: A new performance metric for assessing interconnect designs for extreme-scale HPC systems. In *Proc. of IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS, pages 273–282, Washington, DC, USA, May 2014. IEEE Computer Society.