DECO: Joint Computation, Caching and Forwarding in Data-Centric Computing Networks

Khashayar Kamran
Northeastern University
kamrank@ece.neu.edu

Edmund Yeh
Northeastern University
eyeh@ece.neu.edu

Qian Ma
Northeastern University
qianma@northeastern.edu

ABSTRACT

The emergence of IoT devices and the predicted increase in the number of data-driven and delay-sensitive applications highlight the importance of dispersed computing platforms (e.g. edge computing and fog computing) that can intelligently manage in-network computation and data placement. In this paper, we propose the DECO (Data-cEntric COmputation) framework for joint computation, caching, and request forwarding in data-centric computing networks. DECO utilizes a virtual control plane which operates on the demand rates for computation and data, and an actual plane which handles computation requests, data requests, data objects and computation results in the physical network. We present a throughput optimal policy within the virtual plane, and use it as a basis for adaptive and distributed computation, caching, and request forwarding in the actual plane. We demonstrate the superior performance of the DECO policy in terms of request satisfaction delay as compared with several baseline policies, through extensive numerical simulations over multiple network topologies.

CCS CONCEPTS

• **Networks** → **Network algorithms**; **Network services**; *Network resources allocation.*

KEYWORDS

Distributed computing networks, fog computing, mobile edge computing, data-intensive computing, data-centric computing, caching

ACM Reference Format:
Khashayar Kamran, Edmund Yeh, and Qian Ma. 2019. DECO: Joint Computation, Caching and Forwarding in Data-Centric Computing Networks. In *The Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing (Mobihoc '19), July 2–5, 2019, Catania, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3323679.3326509

1 INTRODUCTION

Centralized clouds have dominated IT service delivery over the past decade. Operating over the internet, and the clouds' low cost of operation [20] has made them the primary means of achieving energy efficiency and computation speed-up for resource-poor devices [1, 7]. Computation offloading to the cloud for mobile users has been studied extensively in the literature. Notable software platforms for computational offloading using device and network profiling are introduced in [6, 9, 14, 16]. Analytical models for optimal offloading are also proposed in [22, 24]. Recently, the cost efficiency and scalability of centralized cloud have been challenged by the emergence of Internet of Things (IoT) devices and the predicted increase in services with ultra low latency requirements (one millisecond or less)[19, 20]. This has made paradigms such as fog computing [2] and mobile edge computing more appealing. In this paper we refer to the family of such paradigms as *dispersed computing*.

In the fog computing paradigm, networking, computation and storage resources are distributed at different hierarchical levels from the core of the network to the edge. In mobile edge computing, these resources are distributed throughout the mobile edge close to the users. As the number of delay-sensitive applications increases, these platforms have the potential to outperform centralized cloud architectures in terms of request satisfaction delay [19]. The potential benefits of such paradigms are accompanied by challenges in distributed implementation and control. Another challenge is the increased popularity of media-rich and data-driven applications where computations are often designed to be performed on large pieces of data stored in the network. Medical data analytics [4, 5], data processing for wearable devices [11], intelligent driving and transportation systems [28] and in-network image/video processing [17] are examples of such applications.

A fundamental question in dispersed computing is how to optimally utilize the processing, storage and bandwidth resources in the network to accomplish data-centric computation with high throughput and low latency. Specifically, how should one forward computation requests, perform computations, and move and store data in the network? For instance, should one bring data to the computation-requesting node for computation or take the computation to the data server? How can one provide a solution in a distributed and adaptive[1] manner within networks with general topology and request patterns?

While previous work has addressed aspects of the fundamental question raised above, the problem has never been solved as a coherent whole. To the best of our knowledge, this paper is the first to study joint computation, forwarding, data placement and caching within an adaptive and distributed setting.

In this paper, we consider a data-centric dispersed computing network with arbitrary topology and arbitrary processing, communication and storage resources available at each node. Users issue *computation requests* for performing a computation task on

Mobihoc '19, July 2–5, 2019, Catania, Italy

ACM ISBN 978-1-4503-6764-6/19/07…$15.00
https://doi.org/10.1145/3323679.3326509

[1]By adaptive, we mean that control algorithms do not require knowledge of computation request rates.

a required piece of data. (e.g. processing a 3D map of the environment in AR/VR applications). This computation request, along with input arguments (e.g. the feature map of users' point of view in AR/VR applications), is forwarded through the network until a node decides to perform the task locally. This node can process the computation only when it has the required data object, either from its own cache or fetched from the other nodes by issuing a *data request*. After the data request arrives at a data server or caching point, a copy of the data is sent back to the data requester on the reverse path. Each node on the path can optionally cache the data for future use. The data requester then processes the task and sends the result back to the original computation requester on the reverse path (of the computation request). The question is how nodes should decide on computation request forwarding, data request forwarding, computation and caching in an adaptive and distributed manner.

To solve this challenging problem, we propose the DECO (DataCEntric COmputation) framework. DECO utilizes two metrics, called *virtual computation interest (VCI)* and *virtual data interest (VDI)*, to capture the measured demand for computations and data objects, respectively. Virtual interest as a metric for measured demand was first introduced in [25] for caching networks where a *virtual plane* was employed to handle the virtual interests. Similarly, here we deploy a virtual plane that operates on the VCIs and VDIs, and an *actual plane* which handles actual computation requests, data requests, data objects and computation results in the physical network. This separation between virtual and actual plane allows us to design the elegant DECO joint computation/caching/forwarding algorithm. The DECO algorithm is proved to be throughput optimal in the virtual plane, and is used in the actual plane to decide on forwarding, computation and caching actions in the network. The superior performance of the DECO algorithm, relative to many baseline algorithms, is shown through extensive simulation studies.

Our key contributions in this work can be summarized as follows:

- We present an adaptive and distributed framework called DECO for joint computation, caching and request forwarding in a data-centric dispersed computing network with arbitrary topology, data catalog and task catalog consisting of single-stage computations.
- The proposed DECO framework consists of a virtual control plane and an actual plane. This allows us to design an elegant algorithm which is shown to be throughput optimal in the virtual plane, and is used for decision making in the actual plane. The throughput optimal algorithm takes into account demand for both computation and data to optimize computation, caching, and forwarding decisions.
- We present new proofs for the stability region and throughput optimality of the DECO policy. This is necessitated in part by new dynamics caused by internally generated data demand from computation requests.
- We evaluate the performance of the DECO framework in terms of computation request satisfaction latency through extensive simulations. We show that the DECO solution significantly outperforms a number o baseline schemes over all network topologies tested.

2 RELATED WORK

Task scheduling and resource allocation in a heterogeneous computing platform have been studied in research and practice. Authors in [21] studied the problem of static task scheduling and proposed a centralized heuristic called HEFT for scheduling tasks represented by a DAG (Directed Acyclic Graph). Pegasus [10] and CIRCE [15] were proposed as frameworks for mapping of tasks to computers in a dispersed computing system. Distributed assignment of tasks to computers was studied in [18]. These works present frameworks for task assignment problem but do not provide any optimality guarantee on the performance of the computing network.

The problem of virtual function placement in a computing network was studied in [8] where the objective is to minimize the cost of setting up functions and requesting service. Authors in [12] studied a similar problem in a distributed cloud network where computation services are modeled as a chain of consecutive tasks, and there is a linear cost associated with setting up and utilizing computation and communication resources. They propose a throughput optimal method for task scheduling and request forwarding based on minimization of Lyapunov drift plus penalty to minimize the cost while keeping the computation queues stable. A throughput optimal policy for uni-cast and multi-cast flows was proposed in [27] based on a layered graph model. A throughput optimal policy for more general DAG-based service models was proposed in [23].

Although these methods stabilize the computation queues, in a data-centric computing network the solution should also take stabilization of data queues into account. Authors in [3] proposed a solution to joint caching and computation at the mobile edge where caching is used to store the final result of computations. Similarly, authors in [26] studied the problem of task scheduling and image placement in order to minimize the request satisfaction delay. In contrast to our work, these works study specific one-hop and two-hop topologies in a centralized fashion where request rates are known in prior.

3 COMPUTATION NETWORK MODEL

Consider a network of computing nodes, each capable of processing computation tasks, caching data objects and communicating with other computing nodes in the network. We model the network as a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with $\mathcal{V}$ and $\mathcal{E}$ representing network nodes and links respectively. Assume that $(b, a) \in \mathcal{E}$ whenever $(a, b) \in \mathcal{E}$. Each node $v \in \mathcal{V}$ is equipped with a processor with capacity of P_v (in instructions per second) and a cache with capacity of C_v (in bits). We let the transmission capacity on link $(a, b) \in \mathcal{E}$ be C_{ab} (in bits per second). There is a set $\mathcal{F}$ of tasks which all nodes are capable of processing. These computation tasks operate on a set $\mathcal{D}$ of data objects, each stored somewhere in the network. A user interested in computation services issue a computation request from the set of available requests $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{D}$. Each requests for performing the mth task on the kth data object is associated with unique user-specified inputs with negligible size compared to the required data object k. We assume computation load and size of the results are determined by the computation task m and the data object k, and not by the user specified inputs. Thus we specify a computation request in the network by a pair $(m, k) \in \mathcal{R}$. Assume the size of the kth data object to be L_k (in bits) and the

size of the result of mth task on kth data object to be $Z_{(m,k)}$ (in bits). The computation load of performing mth task on kth data object is denoted by $q_{(m,k)}$ (in number of instructions). We assume for each data object $k \in \mathcal{D}$ there is a designated node denoted by $src(k) \in \mathcal{V}$ which serves as the permanent source of the data object [2]. At the source nodes, the cache space is additional to the storage space needed for permanently stored data objects. Since requests of type (m,k) require data object k for processing, $src(k)$ can always process such computation requests and therefore can serve as a computing node for these requests.

We assume routing information [3] to the source nodes is already populated in every node. Requests of type (m,k) arrive at node $n \in \mathcal{V}$ according to a stationary and ergodic process $A_n^{(m,k)}(t)$ with average arrival rate $\lambda_n^{(m,k)} \triangleq \lim_{t\to\infty} \frac{1}{t}\sum_{\tau=1}^{t} A_n^{(m,k)}(\tau)$. A node receiving a computation request generates a *computation interest packet* with negligible size (compared to the data objects and the computation results' size) containing the task identification (m), data identification (k), and input arguments to be forwarded through the network. Each node receiving a computation interest packet decides to whether or not *perform* the computation request locally. In this paper we differentiate between *performing computation request* and *processing computation request*. The difference is explained in the procedure below:

1: **procedure** PERFORMING COMPUTATION (m, k) AT NODE n
2: **if** data object k is stored at node n **then** send computation request to the processor queue for *processing*.
3: **else**
4: put computation request in *pending computation requests for data object* k $\left(PCR(k)\right)$ queue.
5: issue a request for fetching data object k by creating a *data interest packet*.
6: **if** data object k arrives at node n **then** put the computation requests in the $PCR(k)$ queue into the processor queue in First-Come-First-Served order.

If the node does not decide to perform the computation locally, it can forward the computation interest packets to its neighbors. The receiving neighbor remembers the interface on which it receives the interest packet. The node which processes the computation puts the result into *result packets* and sends it back on the reverse path (of the computation interest packet) to the original requester.

A node issues a *data interest packet* whenever it decides to perform a task but does not have the required data object stored locally. As in the case for computation interest packets, nodes receiving data interest packets remember the incoming interface. When a node receives a data interest packet for an object which is in its cache, it creates a copy of that data object, puts it into *data packets* and sends it back on the reverse path (of the data interest packet) to the requester. Nodes receiving data objects on the reverse path have the option to cache them for future use. A graphical overview of the network can be seen in Figure 1. A graphical representation of procedures discussed above at each node can be seen in Figure 2

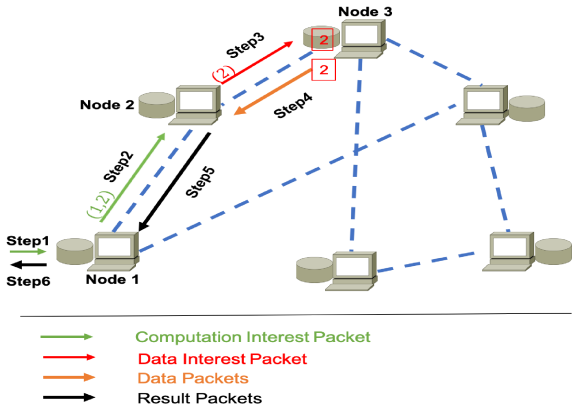


Figure 1: A data-centric computing network. Step 1: A computation request $(1, 2) \in \mathcal{R}$ arrives at the network. Step 2: Node 1 creates a computation interest packet (green) and forwards it to the source of data object 2 (Node 3). Step 3: Node 2 receives the computation interest packet and decides to perform it locally. Since it does not have data object 2 stored in the cache, it puts the computation request in the $PCR(2)$ queue and generates a data interest packet (red) for and forwards it toward the source (Node 3). Step 4: Node 3 receives a data interest packet for data object 2. It creates a copy of the data object 2 and forwards it on the reverse path toward the requester of data (Node 2). Step 5: Node 2 receives data object 2 and sends the pending computation request $(1, 2)$ to the processor. Once it is processed, Node 2 sends the result on the reverse path to the original requester (Node 1). Step 6: Node1 delivers the result to the user.

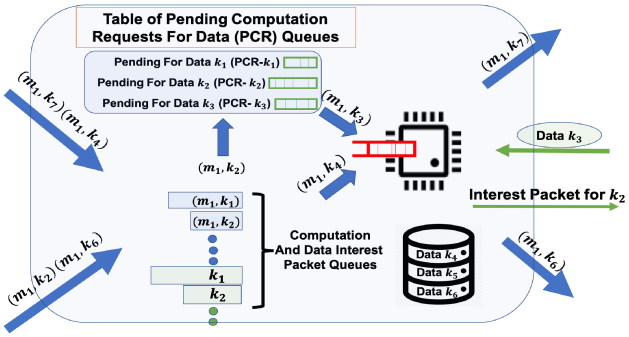


Figure 2: Performing computation, forwarding and caching at nodes. Computation interest packets for (m_1, k_2), (m_1, k_4), (m_1, k_6), (m_1, k_7) are forwarded to the node. The node decides to perform (m_1, k_2) and (m_1, k_4) and forwards (m_1, k_7) and (m_1, k_6) to its neighbors. (m_1, k_4) is sent directly to the processor queue since k_4 is already stored in the cache. Since k_2 is not available in the cache, request (m_1, k_2) is put in the $PCR(k_2)$ queue and a data interest packet for k_2 is generated and forwarded. At this time, k_3 arrives to the node and it sends all pending computation requests for k_3 (e.g. (m_1, k_3)) to the processor.

There are several problems needed to be solved in this setting. Specifically, how to forward the computation and data interest

[2] This setting can be extended to a scenario where there are multiple designated sources for each data object.
[3] We want to point out that routing and forwarding are two different procedures. Routing is a network-wide process that provides possible forwarding interfaces toward the destination at each node and forwarding is the action of transferring packets to appropriate output interface.

packets, how to decide on performing computations and caching, and how to make these decisions in a distributed and scalable fashion. In the next section we present DECO framework as a solution to this multi-dimensional problem.

Table 1: Notation Summary

$\mathcal{G}(\mathcal{V}, \mathcal{E})$	Network graph with nodes $\mathcal{V}$ and edges $\mathcal{E}$
$\mathcal{F}$	Catalog of available tasks
$\mathcal{D}$	Catalog of available data objects
$\mathcal{R}$	Set of available computation requests ($\mathcal{R} \subseteq \mathcal{F} \times \mathcal{D}$)
$PCR(k)$	The queue of pending computation requests for data object k at each node
P_v	Processor capacity at node $v \in \mathcal{V}$
C_v	Cache capacity at node $v \in \mathcal{V}$
C_{ab}	Transmission capacity of link $(a, b) \in \mathcal{E}$
L_k	Size of data object $k \in \mathcal{D}$
$Z_{(m,k)}$	Size of the result size for computation request $(m, k) \in \mathcal{R}$
$q_{(m,k)}$	Computation load of computation request $(m, k) \in \mathcal{R}$
$\lambda_n^{(m,k)}$	Average arrival rate of computation request $(m, k) \in \mathcal{R}$
$Y_n^{(m,k)}(t)$	Virtual computation interest (VCI) count for $(m, k) \in \mathcal{R}$ in node n at the beginning of time slot t
$V_n^k(t)$	Virtual data interest (VDI) count for $k \in \mathcal{D}$ in node n at the beginning of time slot t
$A_n^{(m,k)}(t)$	Number of exogenous arrivals at node n for computation request (m, k) during time slot t
$\mu_{ab}^{(m,k)}(t)$	Allocated transmission rate of VCIs for (m, k) on link (a, b) during time slot t
$\nu_{ab}^{(m,k)}(t)$	Allocated transmission rate of VDIs for $k \in \mathcal{D}$ on link (a, b) during time slot t
$\mu_{n,proc}^{(m,k)}(t)$	Allocated processing rate of VCIs for (m, k) at node n during time slot t
$s_n^k(t)$	Caching state for object k at node n during slot t in the virtual plane

4 DECO FRAMEWORK

In this section we introduce DECO framework for joint computation, forwarding and caching in the setting discussed in section 3. DECO relies on two metrics called *virtual computation interest (VCI)* and *virtual data interest (VDI)*. VCIs and VDIs are counts tracked by each node and capture the measured demand for computations and data objects respectively. Specifically, the virtual interest counts for a computation or a data object in a given part of the network represent the local level of interest in that computation or data object, as determined by network topology, users' demand and the capability of nodes in satisfying this demand through caches and processors. As illustrated in Figure 3, the VCIs and VDIs are handled by a *virtual control plane* which sits on top of the actual plane handling actual computation requests, data requests, data objects and results. This separation allows to formulate an elegant algorithm to be discussed in section 4.1.

In what follows we discuss the dynamic through which virtual interests are created and handled in the virtual plane. We characterize the stability region in the virtual plane, and we present a throughput optimal control policy within the virtual plane that adaptively stabilizes all VCI and VDI queues for all computation request arrival rates inside the stability region. Based on the throuhgput optimal policy, we design a method for joint computation, request forwarding and caching in the actual plane that significantly outperforms a number of baseline schemes as we show in section 5.

4.1 Virtual Plane Dynamics

Here we describe the dynamics of virtual interests within the virtual plane. Consider time slots of length 1 (without loss of generality)

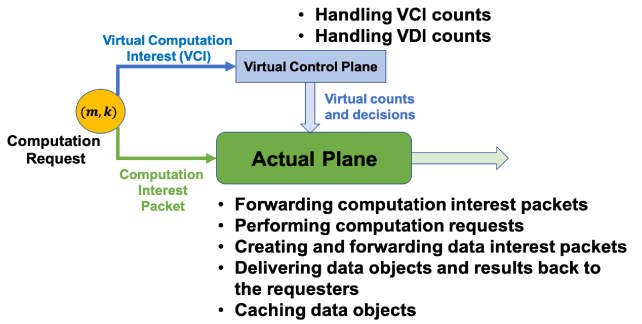


Figure 3: DECO framework. Virtual plane on top of the actual plane.

indexed by $t = 1, 2, \ldots$, where time slot t refers to the interval $[t, t+1)$. Each node $n \in \mathcal{V}$ keeps a separate queue for VCIs corresponding to the request $(m, k) \in \mathcal{R}$. The count of this queue at the beginning of time slot t is denoted by $Y_n^{(m,k)}(t)$. Each node also keeps separate queues for VDIs corresponding to data object $k \in \mathcal{D}$ and its count is denoted by $V_n^k(t)$. Initially all VCI and VDI queues are empty, i.e., $Y_n^{(m,k)}(1) = V_n^k(1) = 0 \quad for\ all\ n, m, k$. For each computation request (m, k) entering the network, the count $Y_n^{(m,k)}$ is increased accordingly at the entry nodes. In the virtual plane, we assume that at each time slot t, each node $n \in \mathcal{V}$ can access to any data object $k \in \mathcal{D}$. Thus, nodes can process any VCI or cache any data object in the virtual plane without waiting for the data to be fetched. This assumption enables us to design an elegant algorithm in the virtual plane. Based on this assumption, processors and caches act as sinks for the VCIs and the VDIs respectively. Nodes decrease their VCI counts by processing them in the virtual plane and decrease their VDI counts by caching the corresponding data object in the virtual plane. On the other hand, processing a computation task in a node results in the increased local demand for the required data object in that node. In order to capture this, processing VCIs in the virtual plane leads to increase in the VDI counts for the corresponding data objects. Nodes can also forward VCIs and VDIs to their neighbors.

Let $A_n^{(m,k)}(t)$ be the number of exogenous computation request arrivals at node n for computation (m, k) during time slot t. For every computation request (m, k) arriving at node n, a corresponding VCI for (m, k) is generated at n ($Y_n^{(m,k)}(t)$ incremented by 1). The long term exogenous arrival rate at node n for computation (m, k) is $\lambda_n^{(m,k)} \triangleq \lim_{t\to\infty} \frac{1}{t} \sum_{\tau=1}^{t} A_n^{(m,k)}(\tau)$.

Let $\mu_{ab}^{(m,k)}(t)$ be the allocated transmission rate of VCIs for (m, k) on link (a, b) during time slot t. Also, Let $\nu_{ab}^k(t)$ be the allocated transmission rate of VDIs for data object k on link (a, b) during time slot t. Note that at each time slot, a single message between node a and node b can summarize all virtual interest transmissions during time slot t. We denote allocated processing rate of VCIs for (m, k) at node n during time slot t by $\mu_{n,proc}^{(m,k)}(t)$. As we mentioned before, in order to capture the local demand for data objects, for each VCI that is processed (i.e. its count is decreased by 1), a VDI is generated (i.e. its count is increased by 1). Let $s_n^k(t) \in \{0, 1\}$ represent the caching state for object k at node n during slot t in the virtual plane, where

$s_n^k(t) = 1$ if object k is cached at node n during slot t, and $s_n^k(t) = 0$ otherwise. Now note that even if $s_n^k(t) = 1$, the cache at node n can satisfy only a limited number of interests during one time slot. This is because there is a maximum rate r_n (in objects per slot) at which node n can produce copies of cached object k. These dynamics can be written in details as follows:

$$Y_n^{(m,k)}(t+1) \leq \left(Y_n^{(m,k)}(t) - \sum_{b\in\mathcal{V}} \mu_{nb}^{(m,k)}(t) - \mu_{n,proc}^{(m,k)}(t)\right)^+ + \sum_{a\in\mathcal{V}} \mu_{an}^{(m,k)}(t) + A_n^{(m,k)}(t) \quad (1)$$

$$V_n^k(t+1) \leq \left(V_n^k(t) - \sum_{b\in\mathcal{V}} v_{nb}^k(t) - r_n s_n^k(t)\right)^+ + \sum_{m\in\mathcal{F}} \mu_{n,proc}^{(m,k)}(t) + \sum_{a\in\mathcal{V}} v_{an}^k(t) \quad (2)$$

where $(x)^+ \triangleq max\{x, 0\}$. Also, $V_{src(k)}^k(t) = 0 \quad for\ all\ t \geq 1$ and , $Y_{src(k)}^{(m,k)} = 0 \quad for\ all\ m \in \mathcal{F}, for\ all\ t \geq 1$. A graphical representation of dynamics in the virtual plane can be seen in Figure 4.

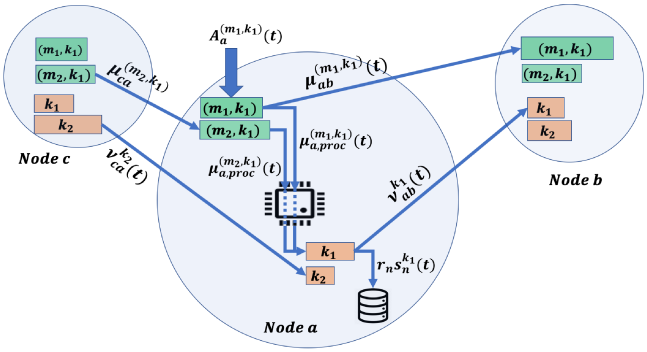


Figure 4: Virtual plane dynamics at node a. VCI and VDI queues evolving according to (1)-(2).

From (1) and (2), it can be seen that the VCIs for (m, k) are processed and decreased with rate $\mu_{n,proc}^{(m,k)}(t)$ and VDIs for data object k are decreased with rate r_n if node n decides to cache the data object k in the virtual plane ($s_n^k(t) = 1$). If there are any VCI or VDI left in the node, they are transmitted to the neighbors with rate $\sum_b \mu_{nb}^{(m,k)}(t)$ and $\sum_b v_{nb}^k(t)$ respectively. The exogenous arrivals $A_n^{(m,k)}(t)$ and endogenous arrivals $\sum_a \mu_{an}^{(m,k)}(t)$ during time slot t is added to the VCI queue at the end of time slot. The number of VCIs processed corresponding to data object k and the endogenous arrivals $\sum_a v_{an}^k(t)$ during time slot t is added to the VDI queue at the end of time slot. Note that (1) is a inequality since the number of VCIs for (m, k) arriving to node n during slot t may be less than $\sum_a \mu_{an}^{(m,k)}(t)$ if the neighboring nodes have little or no VCI to transmit. Also (2) is inequality because the number of VDIs for object k arriving to node n during slot t may be less than $\sum_b v_{nb}^k(t)$ and the number of VDIs created due to the processing of VCIs might be less than $\sum_m \mu_{n,proc}^{(m,k)}(t)$ if node n has little or no VCI to process.

4.2 Throughput Optimal Policy in the Virtual Plane

In this section we introduce a distributed policy based on Lyapunov drift minimization for throughput optimal decision making in the virtual plane. The drift minimization problem results in two different LP problems for allocating processing and transmission rates. It also involves solving a knapsack problem for caching decision which is NP-hard in general, but can be solved efficiently using approximation techniques or dynamic programming at each node. As we will discuss shortly, in the settings where the size of data objects are equal and/or the size of results are equal and/or the computation loads are equal, this joint problem is simpler to solve. We then introduce the stability region for the virtual interests and prove that the proposed joint policy is throughput optimal within the virtual plane. Finally, we design a distributed policy for computation, forwarding and caching in the actual plane based on the optimal policy obtained in the virtual plane.

In what follows we introduce the joint processing, transmission and caching policy in the virtual plane:

Algorithm 1. *At the beginning of each time slot t, observe the counts for VCIs $(Y_n^{(m,k)}(t))_{n\in\mathcal{V},(m,k)\in\mathcal{R}}$ and VDIs $(V_n^k(t))_{n\in\mathcal{V},k\in\mathcal{D}}$ and decide on processing, transmission and caching in the virtual plane as follows.*

Processing: *at each node n, choose processing rates of VCIs by solving the following LP:*

$$maximize \sum_{(m,k)\in\mathcal{R}} \mu_{n,proc}^{(m,k)}(t)\left(Y_n^{(m,k)}(t) - V_n^k(t)\right) \quad (3)$$

$$subject\ to \sum_{(m,k)\in\mathcal{R}} q_{(m,k)}\mu_{n,proc}^{(m,k)}(t) \leq P_n \quad (4)$$

Transmission: *at each node n, choose transmission rate of VCIs and VDIs by solving the following LP:*

$$maximize \sum_{(m,k)\in\mathcal{R}} \mu_{nb}^{(m,k)}(t)\left(Y_n^{(m,k)}(t) - Y_b^{(m,k)}(t)\right) + \sum_{k\in\mathcal{D}} v_{nb}^k(t)\left(V_n^k(t) - V_b^k(t)\right) \quad (5)$$

$$subject\ to \sum_{k\in\mathcal{D}} L_k v_{nb}^k(t) + \sum_{(m,k)\in\mathcal{R}} Z_{(m,k)}\mu_{nb}^{(m,k)}(t) \leq C_{bn} \quad (6)$$

Caching: *at each node n, choose caching variables by solving the following knapsack problem:*

$$maximize \sum_{k\in\mathcal{D}} V_n^k(t)s_n^k(t) \quad (7)$$

$$subject\ to \sum_{k\in\mathcal{D}} L_k s_n^k(t) \leq C_n \quad (8)$$

An important aspect of Algorithm 1 is it being distributed. It can be seen that processing and caching decisions are solved at each node separately and each node needs to exchange the VCI and VDI counts only with its own neighbors in the transmission decision problem.

It is worth noting that in a network with equal-sized data, the knapsack caching problem (7)-(8) reduces to a max-weight problem which is solvable in linear time at each node. In a scenario

where computation result sizes are also equal, the LP for transmission rates (5)-(6) turns into a backpressure algorithm on each link. Finally in a scenario where computation loads are equal, the LP problem for processing rates (3)-(4) turns into a backpressure-like algorithm between VCI and VDI queues at each node. Consider a network where all data sizes are equal ($L_k = L \quad \forall k \in \mathcal{D}$), all result sizes are equal ($Z_{(m,k)} = Z \quad \forall (m,k) \in \mathcal{R}$) and all computation loads are equal ($q_{(m,k)} = q \quad \forall (m,k) \in \mathcal{R}$). We can show that in this situation algorithm 1 is reduced to a simple backpressure and sorting algorithm.

Algorithm 2. *In a network with $L_k = L$, $Z_{(m,k)} = Z$, $q_{(m,k)} = q$, at the beginning of each time slot t, observe the counts for VCIs $(Y_n^{(m,k)}(t))_{n\in\mathcal{V},(m,k)\in\mathcal{R}}$ and VDIs $(V_n^k(t))_{n\in\mathcal{V},k\in\mathcal{D}}$ and decide on processing, forwarding and caching in the virtual plane as follows.*

Processing: *at each node n, for each $(m,k) \in \mathcal{R}$ choose:*

$$\mu_{n,proc}^{(m,k)}(t) = \begin{cases} \frac{P_n}{q} & W_{n,proc}^*(t) > 0 \quad and \quad (m,k) = (m,k)^* \\ 0 & otherwise. \end{cases} \tag{9}$$

$$W_{n,proc}^{(m,k)}(t) = Y_n^{(m,k)}(t) - V_n^k(t) \tag{10}$$
$$(m,k)^* = \underset{(m,k)}{argmax} \quad W_{n,proc}^{(m,k)}(t)$$
$$W_{n,proc}^*(t) = (W_{n,proc}^{(m,k)^*}(t))^+$$

Transmission: *at each node n, for each $(m,k) \in \mathcal{R}$ and each $k \in \mathcal{D}$ choose:*

$$\mu_{nb}^{(m,k)}(t) = \begin{cases} \frac{c_{bn}}{Z} & \frac{W_{nb}^*(t)}{Z} \geq \frac{G_{nb}^*(t)}{L}, W_{nb}^*(t) > 0, (m,k) = (m,k)^{**} \\ 0 & otherwise. \end{cases} \tag{11}$$

$$\nu_{nb}^k(t) = \begin{cases} \frac{c_{bn}}{L} & \frac{G_{nb}^*(t)}{L} > \frac{W_{nb}^*(t)}{Z}, G_{nb}^*(t) > 0, k = k^* \\ 0 & otherwise. \end{cases}$$

$$W_{nb}^{(m,k)}(t) = Y_n^{(m,k)}(t) - Y_b^{(m,k)}(t), \quad G_{nb}^k(t) = V_n^k(t) - V_b^k(t) \tag{12}$$

$$(m,k)^{**} = \underset{(m,k)}{argmax} \quad W_{nb}^{(m,k)}(t), \quad k^* = \underset{k}{argmax} \quad G_{nb}^k(t)$$
$$W_{nb}^*(t) = (W_{nb}^{(m,k)^*}(t))^+, \quad G_{nb}^*(t) = (G_{nb}^{k^*}(t))^+$$

Caching: *at each node n, let $(d_1, d_2, \ldots, d_K)$ be a permutation of $\mathcal{D}$ such that $V_n^{d_1} \geq V_n^{d_2} \geq \cdots \geq V_n^{d_K}$ let $i_n = \lfloor \frac{C_n}{L} \rfloor$, then choose*

$$s_n^k(t) = \begin{cases} 1 & k \in \{k_1, k_2, \ldots, k_{i_n}\} \\ 0 & otherwise. \end{cases}$$

In Algorithm 2, each node n at each time t allocates the entire normalized processor capacity ($\frac{P_n}{q}$) to process the VCIs of $(m,k)^*$ which has the maximum difference between its VCI count and VDI count for the required data object k as shown in (9)-(10). The intuition behind this is important. The optimal policy allocates the processing capacity to the computation request for which there is relatively high local demand (i.e. large VCI count) and low local demand for the required data object (i.e. low VDI count, often due to the data object being cached in close vicinity).

Each node n at each time t for transmission on any outgoing link $(n,b) \in \mathcal{E}$ chooses the VCI or VDI that has the maximum backlog difference on the link normalized by size (Z for VCIs and L for VDIs) and allocates the entire normalized reverse link capacity (normalized by Z if the chosen count is a VCI and by L if the chosen count is a VDI) to it as shown in (11)-(12). As for caching, each node n with capacity to cache $i_n = \lfloor \frac{C_n}{L} \rfloor$ data objects, chooses i_n data objects with highest VDI counts to be cached in the virtual plane. We note that in Algorithm 2, at each node the computational complexity is $O(|\mathcal{F}|\times|\mathcal{D}|)$ for processing policy, $O(|\mathcal{V}|\times|\mathcal{F}|\times|\mathcal{D}|)$ for transmission policy and $O(|\mathcal{D}|)$ for caching policy.

In the following section we show that algorithm 1 maximizes the throughput in the virtual plane discussed in section 4 with proper constraints on processing rates, transmission rates and caches.

4.2.1 Virtual Plane Stability Region. Here we show that algorithm 1 maximizes the throughput of virtual plane in the network $\mathcal{G}(\mathcal{V},\mathcal{E})$ with appropriate constraints. We assume

- Exogenous computation request arrival processes (which are also VCI arrival processes as mentioned before) $\{A_n^{(m,k)}(t); t = 1, 2, \ldots\}$ are mutually independent with respect to $n, (m,k)$.
- $\{A_n^{(m,k)}(t); t = 1, 2, \ldots\}$ are i.i.d with respect to t for all $n \in \mathcal{V}, (m,k) \in \mathcal{R}$
- For all $n \in \mathcal{V}, (m,k) \in \mathcal{R}$, $A_n^{(m,k)}(t) \leq A_{n,max}^{(m,k)}$ for all t where $A_{n,max}^{(m,k)} \in \mathbb{R}_+$.

As for the constraints, during each time slot a node cannot store more than its cache capacity and cannot process computation requests more than processor capacity. For each computation interest packet sent on a link, a result comes back on the reverse link eventually and for each data interest packet sent on a link, a data object traverses back on the reverse link. Since we assume the size of interest packets are negligible compared to results and data objects, when sending interest packets on a link (a,b) we need to take into account the reverse link capacity. These constraints should be reflected in the virtual plane and can be summarized as follows:

$$\sum_{k\in\mathcal{D}} L_k s_n^k(t) \leq C_n \quad \forall n \in \mathcal{V} \tag{13}$$

$$\sum_{(m,k)\in\mathcal{R}} q_{(m,k)} \mu_{n,proc}^{(m,k)}(t) \leq P_n \quad \forall n \in \mathcal{V} \tag{14}$$

$$\sum_{k\in\mathcal{D}} L_k \nu_{ab}^k(t) + \sum_{(m,k)} Z_{(m,k)} \mu_{ab}^{(m,k)}(t) \leq C_{ba} \quad \forall (a,b) \in \mathcal{E} \tag{15}$$

In order to show the throughput optimality, we present the virtual plane stability region in this section. Stability for VCI and VDI queues at node n is defined as:

$$\limsup_{t\to\infty} \frac{1}{t} \sum_{\tau=1}^{t} 1_{[Y_n^{(m,k)}(\tau) > \xi]} \to 0 \text{ as } \xi \to \infty$$

$$\limsup_{t\to\infty} \frac{1}{t} \sum_{\tau=1}^{t} 1_{[V_n^k(\tau) > \xi]} \to 0 \text{ as } \xi \to \infty$$

where $1_{\{.\}}$ is the indicator function. The stability region Λ is the is the closure of the set of all computation arrival rates defined

as $\lambda_n^{(m,k)} \triangleq \lim_{t\to\infty} \frac{1}{t}\sum_{\tau=1}^{t} A_n^{(m,k)}(\tau)$ for which there exists some feasible processing, forwarding, and caching policy which can stabilize all VCI and VDI queues. By feasible, we mean that at any time t, the caching vectors $\left(s_n^k(t)\right)_{k\in\mathcal{D},n\in\mathcal{V}}$ satisfy (13), the processing rate vector $\left(\mu_{n,proc}^{(m,k)}\right)_{(m,k)\in\mathcal{R},n\in\mathcal{V}}$ satisfy (14) and the forwarding rate vector $\left(\mu_{ab}^{(m,k)}\right)_{(m,k)\in\mathcal{R},(a,b)\in\mathcal{E}}$ satisfy transmission constraints in (15). The following theorem characterizes the stability region in the virtual plane:

Theorem 1. *The stability region for virtual computation and data interests of the network $\mathcal{G}(\mathcal{V},\mathcal{E})$ with caching, computation and transmission capacity constraints given by (13)-(14)-(15) and queue dynamics (1)-(2), is the set Λ consisting of all computation request arrival rates $\lambda_n^{(m,k)}$ such that there exists computation flow variables $(f_{ab}^{(m,k)})_{(m,k)\in\mathcal{R},(a,b)\in\mathcal{L}}$, data flow variables $(d_{ab}^k)_{k\in\mathcal{D},(a,b)\in\mathcal{L}}$, processing flow variables $(f_{n,proc}^{(m,k)})_{(m,k)\in\mathcal{R},n\in\mathcal{N}}$ and caching variables $(\beta_{n,i})_{n\in\mathcal{N};i\in\Psi_n}$ satisfying*

$$f_{ab}^{(m,k)} \geq 0, f_{nn}^{(m,k)} = 0, f_{src(k)n}^{(m,k)} = 0 \forall a,b,n\in\mathcal{N},(m,k)\in\mathcal{R} \quad (16)$$

$$f_{ab}^{(m,k)} = 0 \quad \forall a,b\in\mathcal{N},(m,k)\in\mathcal{R},(a,b)\notin\mathcal{L} \quad (17)$$

$$d_{ab}^k \geq 0, d_{nn}^k = 0, d_{src(k)n}^k \quad \forall a,b,n\in\mathcal{N},k\in\mathcal{D} \quad (18)$$

$$d_{ab}^k = 0 \quad \forall a,b\in\mathcal{N},k\in\mathcal{D},(a,b)\notin\mathcal{L} \quad (19)$$

$$0 \leq \beta_{n,i} \leq 1 \quad i\in\Psi_n \quad (20)$$

$$f_{n,proc}^{(m,k)} \geq 0 \quad \forall n\in\mathcal{N},(m,k)\in\mathcal{R} \quad (21)$$

$$\lambda_n^{(m,k)} \leq \sum_{b\in\mathcal{V}} f_{nb}^{(m,k)} - \sum_{a\in\mathcal{V}} f_{an}^{(m,k)} + f_{n,proc}^{(m,k)} \quad \forall n\in\mathcal{N},(m,k)\in\mathcal{R} \quad (22)$$

$$\sum_{a\in\mathcal{V}} d_{an}^k + \sum_m f_{n,proc}^{(m,k)} \leq \sum_{b\in\mathcal{V}} d_{nb}^k + r_n \sum_{i\in\Psi_n} \beta_{n,i}\mathbf{1}[k\in\mathcal{B}_{n,i}] \quad \forall n\in\mathcal{N},(m,k)\in\mathcal{R} \quad (23)$$

$$\sum_{(m,k)\in\mathcal{R}} Z_{(m,k)} f_{ab}^{(m,k)} + \sum_{k\in\mathcal{D}} L_k d_{ab}^k \leq C_{ba} \quad \forall (a,b)\in\mathcal{L} \quad (24)$$

$$\sum_{i\in\Psi_n} \beta_{n,i} = 1 \quad \forall n\in\mathcal{N} \quad (25)$$

$$\sum_{(m,k)\in\mathcal{R}} q_{(m,k)} f_{n,proc}^{(m,k)} \leq P_n \quad \forall n\in\mathcal{N} \quad (26)$$

Where Ψ_n is the set of feasible cache combination for node n.

Proof. See Appendix A of the technical report in [13]. □

To our knowledge, Theorem 1 is the first description of the stability region of a data-centric computing network that incorporates the effect of computation, transmission and caching all together.

4.2.2 Throughput Optimality. We now show that Algorithm 1 stabilizes all VCI and VDI queues in the network for any $\boldsymbol{\lambda}\in int(\Lambda)$, without any knowledge of $\boldsymbol{\lambda}$. As a result Algorithm 1 is throughput optimal in the sense of adaptively maximizing the throughput of virtual computation interests.

Theorem 2. *(Throughput Optimality) If there exists $\boldsymbol{\epsilon} = (\epsilon_n^{(m,k)})_{n\in\mathcal{V},(m,k)\in\mathcal{R}} \succ \mathbf{0}$ such that $\boldsymbol{\lambda}+\boldsymbol{\epsilon}\in\Lambda$, then there exists $(\epsilon_n^{(m,k)'})_{n\in\mathcal{V},(m,k)\in\mathcal{R}}, (\epsilon_n^{k'})_{n\in\mathcal{V},k\in\mathcal{D}} \succ \mathbf{0}$ such that the network of virtual interest queues under algorithm 1 satisfies:*

$$\limsup_{t\to\infty} \frac{1}{t}\sum_{\tau=1}^{t}\Big(\sum_{n,(m,k)} E[Y_n^{(m,k)}(\tau)] + \sum_{n,k} E[V_n^k(\tau)]\Big) \leq \frac{NB}{\epsilon'} \quad (27)$$

where $B \triangleq \frac{1}{2N}\sum_{n\in\mathcal{V}}(\mu_{n,out}^{max}+\mu_{n,proc}^{max}+r_n^{max})^2+(\mu_{n,in}^{max}+\mu_{n,proc}^{max}+A_n^{max})^2$, $\epsilon' \triangleq min\{(\epsilon_n^{(m,k)'})_{n\in\mathcal{V},(m,k)\in\mathcal{R}}, (\epsilon_n^{k'})_{n\in\mathcal{V},k\in\mathcal{D}}\}$ $\mu_{n,proc}^{max} \triangleq \frac{P_n}{min\{q_{(m,k)}\}}, \mu_{n,out}^{max} \triangleq \frac{\sum_b C_{bn}}{min\{Z_{(m,k)},L_k\}}$, $\mu_{n,in}^{max} \triangleq \frac{\sum_a C_{na}}{min\{Z_{(m,k)},L_k\}}, r_n^{max} \triangleq r_n|\mathcal{D}|, A_n^{max} \triangleq \sum_{(m,k)} A_{n,max}^{(m,k)}$ $(\epsilon_n^{(m,k)'})_{n\in\mathcal{V},(m,k)\in\mathcal{R}}, (\epsilon_n^{k'})_{n\in\mathcal{V},k\in\mathcal{D}}$ are defined in Appendix A of the technical report in [13].

Proof. See Appendix B of the technical report in [13]. □

We wish to point out that the proof is different in nature from the previous proofs for stability, in the sense that ϵ' used in the stability bound in Theorem (2) is a value first introduced in this paper in order to show the stability of VCI and VDI queues. This is due to the internal generation of demand for data in the network (as described in section 4.1), and the existence of ϵ' is proved in Appendix A of [13].

4.3 Computation, Caching, and Request Forwarding in the Actual Plane

Our goal in this section is to design a distributed joint policy for performing computation, request forwarding and caching in the actual plane based on the throughput optimal algorithm we obtained in section 4.2 for the virtual plane.

We keep a separate queue for each $(m,k)\in\mathcal{R}$ and $k\in\mathcal{D}$ at each node in the actual plane of DECO. In contrast to the virtual plane, as described in the section 3 and in Figure. 2, when nodes decide to perform a computation request (m,k), they send the computation to the processor if they are the source of data object k or have k stored in their cache. Otherwise they put the computation request in the $PCR(k)$ queue and issue a data interest packet for k. When k returns to the node, it sends all computation requests in the $PCR(k)$ queue to the processor. As for caching, nodes can only cache data objects when they are traversing back on the reverse path to the data requester.

4.3.1 Performing Computation Requests. At each time slot t, each node n performs computation requests of $(m,k)\in\mathcal{R}$ with rate $\mu_{n,proc}^{(m,k)^*}(t)$ where $\mu_{n,proc}^{(m,k)^*}(t)$ is the optimal processing rates in the virtual plane at node n in time slot t obtained by solving (3),(4). In other words, at each time slot t each node n takes $\mu_{n,proc}^{(m,k)^*}(t)$ computation interest packet of type (m,k) out of its corresponding queue and sends them to the processor if $n = src(k)$ or has the data object k in its cache. Otherwise it puts them in the $PCR(k)$ queue and generates $\mu_{n,proc}^{(m,k)^*}(t)$ data interest packets for data object k. When data object k reaches to the node n on the reverse path (of the data interest packet), node sends all the pending computation requests in the $PCR(k)$ queue to the processor.

4.3.2 Transmission of Computation and Data Interest Packets. At each time slot t, each node n transmits $\mu_{nb}^{(m,k)^*}(t)$ computation

interest packets of request (m,k) and transmits $v_{nb}^{k^*}(t)$ data interest packets of data object k on each outgoing link $(n,b) \in \mathcal{E}$ where $\mu_{nb}^{(m,k)^*}(t)$ and $v_{nb}^{k^*}(t)$ are optimal transmission rates for VCIs and VDIs in the virtual plane at node n in time slot t obtained by solving (5),(6).

4.3.3 Caching Data Objects. As we mentioned, in the actual plane nodes can only cache data objects when they are traversing back on the reverse path to the requester. We noticed that using virtual caching decisions at each time slot directly in the actual plane leads to oscillatory caching behaviour since data objects can get cached or removed from the cache instantly in the virtual plane. Here we propose a method that results in more stable caching behaviour. For a given window size T, let the *cache score* for object k at node n at time t be

$$CS_n^k(t) = \frac{1}{T} \sum_{\tau=t-T+1}^{t} s_n^{k^*}(\tau) V_n^k(\tau) \tag{28}$$

Where $s_n^{k^*}(t)$ is the optimal caching decision for data object k in the virtual plane at node n in time slot t obtained by solving (7)-(8). This cache score averages over the VDI counts for data object k in the time slots at which node n decided to cache k in the virtual plane, over a sliding window of size T prior to time slot t. When a data objects k_{new} travels back to the requester node, each node on the reverse path cache the data object as long as it has space left in its cache. If the cache is full, the node compares the cache score for k_{new} and the set of currently cached data objects $\mathcal{K}_{n,old}$. If all data objects are of equal size, let $k_{min} \in \mathcal{K}_{n,old}$ be a current cached object with the smallest cache score. If k_{new} has a higher cache score than k_{min}, then k_{min} is evicted and replaced with k_{new}. Otherwise, the cache is unchanged. If data objects have different sizes, the optimal set of objects is chosen to maximize the total cache score under the cache space constraint. This is a knapsack problem that can be solved using approximation techniques at each node.

5 NUMERICAL EVALUATION

This section demonstrates our experimental evaluation of DECO framework. The simulations are performed on four different network topologies: the Abilene topology shown in Figure. 5(a), a fog computing topology shown in Figure. 5(b), the GEANT topology shown in Figure. 5(c) and LHC (Large Hadron Collider) topology which is a prominent data-intensive computing network for high energy physics applications shown in Figure. 5(d).

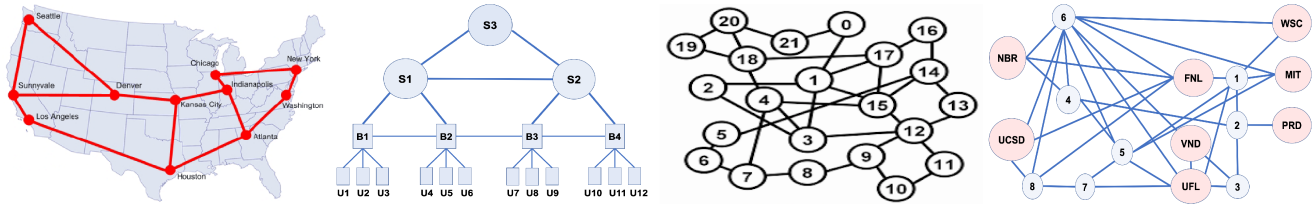
Experiment Setup. In the Abilene topology, the cache capacity is 30GB and the processor capacity is 5×10^5 instructions/sec for all nodes. The link capacity (in both directions) is 240 Gbps for all links. In the Fog topology, the cache capacity is 5GB for $U1, U2, \ldots, U12$ and 25GB for $B1$, $B2$, $B3$, $B4$ and 50GB for $S1$, $S2$, $S3$. The processor capacity is 10^6 instructions/sec for $U1, U2, \ldots, U12$ and 5×10^6 instructions/sec for $B1$, $B2$, $B3$, $B4$ and 10^7 instructions/sec for $S1$, $S2$, $S3$. The link capacity (in both directions) is 40 Gbps for the links between the bottom layer to the second layer $(U1, B1), (U2, B1), \ldots, (U11, B4), (U12, B4)$ and 200 Gbps for $(B1, B2), (B2, B3), (B3, B4), (B1, S1), (B2, S1), (B3, S2), (B4, S2)$ and 400 Gbps for $(S1, S2), (S1, S3), (S2, S3)$. In the GEANT topology, the cache capacity is 30GB and the processor capacity is 25×10^5 instructions/sec for all the nodes. The link capacity (in both directions) is 240 Gbps for all the links. In the LHC topology, for "MIT", "WSC", "PRD", "FNL", "VND", "UFL", "NBR" and "UCSD", the cache capacity is 3TB and processing capacity is 3000, 5000, 5000, 2000, 1000, 1000, 3000, and 2000 instructions/sec respectively. The Cache and processor capacity is zero for all other nodes. The link capacity (in both directions) is 480 Mbps for all links. Other simulation parameters can be seen in Table 2 for each topology. The designated source for each data object is chosen uniformly at random among the source nodes mentioned in Table 2. At each requesting node, computation requests arrive according to a Poisson process with an overall rate λ (in request/node/sec). Each arriving request selects from the set of available tasks (independently) uniformly at random. In the Abilene, Fog and GEANT, we pair the ith computation task with ith data object to form a computation request. In the LHC, we select from the available data objects (independently) according to a Zipf distribution with parameter 1 and pair the selected task and data to form a computation request.

Table 2: Experimental Parameters and Setup

	Abilene	**Fog**	**GEANT**	**LHC**
$\lvert\mathcal{F}\rvert$	100	200	100	100
$\lvert\mathcal{D}\rvert$	100	200	100	500
L_k	3GB	500MB	3GB	60GB
$Z_{(m,k)}$	300MB	50MB	1.5GB	6GB
$q_{(m,k)}$	5×10^4	5×10^4	5×10^4	10^5
Interest Packets' Size	60KB	10KB	60KB	60KB
Source Nodes	Seattle Sunnyvale Los Angles	S3	10, 11, …, 21	MIT, WSC PRD, FNL VND, UFL NBR, UCSD
Requesting Nodes	Atlanta Washington New York	U1, U2, …, U12	0, 1, …, 9	MIT, WSC PRD, FNL VND, UFL NBR, UCSD

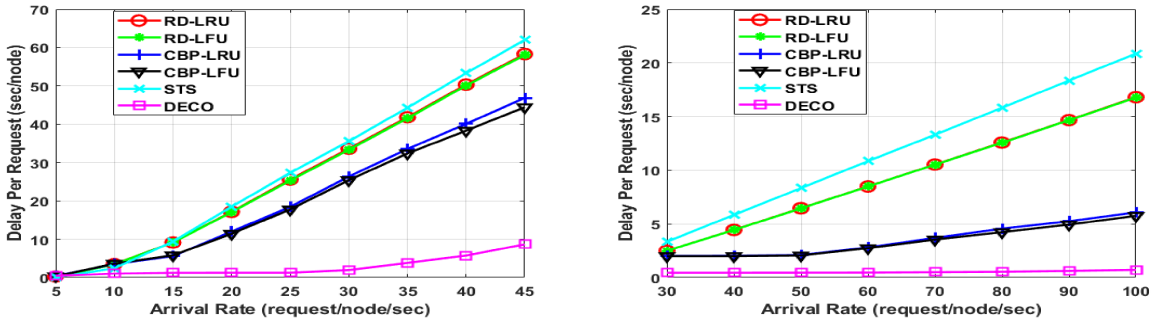
We calculate shortest paths from each node to the source for each data object and populate the forwarding tables of the nodes with this information, beforehand. In all topologies, the buffers holding the computation interest packets, data interest packets, data packets and result packets are assumed to have infinite size. Data packets and result packets share the same queue on the reverse paths and are served on a First-Come-First-Served basis.

Policies and Measurements. We compare DECO with five baseline policies in terms of computation request satisfaction delay. In the **RD-LRU** policy, *RD* stands for "Retrieve Data": Each computation request is performed at the entry node of the request and if necessary, a data interest packets is generated according to the procedure we discussed. All data interest packets in each node share one queue and are forwarded to the source on a First-Come-First-Serve basis. Each node caches the data objects when they travel back on the reverse path to the requesting node and if the cache is full, nodes use *LRU* as cache eviction policy. The **RD-LFU** is similar to the RD-LRU policy but uses *LFU* as its cache

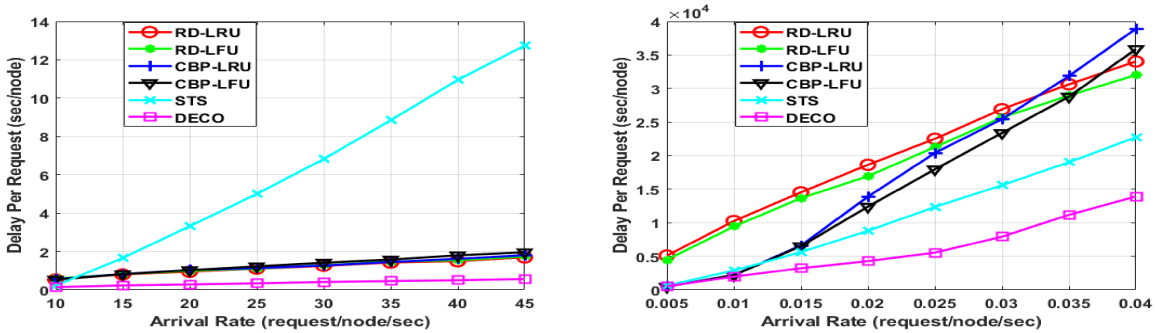


(a) Abilene Topology (b) Fog Topology (c) GEANT Topology (d) LHC Topology

Figure 5: Network topologies



(a) Abilene Topology (b) Fog Topology



(c) GEANT Topology (d) LHC Topology

Figure 6: Computation request satisfaction delay

eviction policy. In the **STS** policy, *STS* stands for "Send To Source" and each computation request (m, k) is forwarded to the source of the data object k. All computation requests share the same queue at each node and are forwarded on a First-Come-First-Serve basis. When the computation requests reach to the source they are sent to the processor queue directly. There is no caching in this policy. In the **CBP-LRU** policy, *CBP* stands for "Computation Backpressure". There is a separate queue for the computation interest packets of type (m, k) at each node. We use backpressure-based algorithms on the computation interest packets for performing computations and forwarding, similar to the approach introduced in [12]. Since all the result sizes and computation loads are equal, the policy performs the most backlogged computation request at each node. Also, the forwarding is done by simple backpressure on each outgoing link subject to the reverse link capacity normalized by the result size. The data interest packets all share the same queue and are forwarded on a First-Come-First-Served basis toward the sources of the data objects. Each node uses *LRU* as its cache eviction policy. The **CBP-LFU** is similar to CBP-LRU policy but uses *LFU* as its cache eviction policy.

The simulator finest granularity *time step* is $2\mu sec$ for the Abilene, Fog and GEANT topology, and is $1msec$ for the LHC topology. In the DECO policy, virtual plane and actual plane decisions are carried out in the slots of length 10^4 time steps and the averaging window size is 10^6 time steps. In the CBP-LRU and CBP-LFU policies, backpressure algorithms for performing computations and forwarding are carried out in the slots of length 10^4 time steps. The average window size in all policies that utilize LFU is 10^6 time steps. The simulator generates computation requests for 100 seconds in the Abilene, Fog and GEANT topology, and 50000 seconds in the LHC topology. After generating requests, simulator waits for all computation requests to be satisfied. The Delay of each computation request is calculated

as the difference between the fulfillment time (i.e., time of arrival of the last result packet) and the creation time of the computation interest packet. We sum over all the delays and divide it by the total number of generated requests and the number of requesting nodes. The computation request satisfaction delay (in second per request per node) is plotted for different arrival rates (in number of requests per node per second) for each topology in Figure 6.

We can see that the DECO policy outperforms all other schemes by a large margin. For instance, at arrival rate of $\lambda = 45$, the DECO has around 80% delay improvement in the Abiline topology and 90% delay improvement in the GEANT topology compared to the closest policy. Another observation is that the second best policy may vary from STS, RD-LFU, or CBP-LFU depending on the size of data objects, computation load, caching, processing and link capacities in each topology. None of the baseline methods is competitive with the DECO, which takes local demand for both computation and data into account for decision making.

6 CONCLUSION

We address the problem of joint computation, caching, and request forwarding in a distributed data-centric computing network where users issue requests for performing a computation task on a piece of data. Our framework utilizes a virtual plane that characterizes the dynamic of demands for computation and data in the networks using virtual computation interest and virtual data interest metrics. We characterize the stability region for the virtual interests and propose a throughput optimal control policy within the virtual plane based on the Lyapunov drift minimization. We show that the optimal policy takes into account virtual data interests as well as virtual computation interests when deciding on computation and forwarding within the virtual plane. By utilizing optimal decisions and counts in the virtual plane, we design a distributed joint request forwarding, computation scheduling and caching policy in the actual plane without any prior knowledge of request arrival rates. Extensive numerical simulations show the superior performance of our method compared to popular baseline policies in terms of computation request satisfaction delay.

ACKNOWLEDGMENTS

This work is supported in part by DARPA grant HR0011-17-C-0050 and National Science Foundation grant NeTS-1718355.

REFERENCES

[1] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. 2013. To offload or not to offload? The bandwidth and energy costs of mobile cloud computing. In *2013 Proceedings IEEE INFOCOM*. 1285–1293.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. ACM, New York, NY, USA, 13–16.

[3] M. Chen, Y. Hao, L. Hu, M. S. Hossain, and A. Ghoneim. 2018. Edge-CoCaCo: Toward Joint Optimization of Computation, Caching, and Communication on Edge Cloud. *IEEE Wireless Communications* 25, 3 (JUNE 2018), 21–27.

[4] M. Chen, Y. Hao, K. Hwang, L. Wang, and L. Wang. 2017. Disease Prediction by Machine Learning Over Big Data From Healthcare Communities. *IEEE Access* 5 (2017), 8869–8879.

[5] M. Chen, Y. Qian, Y. Hao, Y. Li, and J. Song. 2018. Data-Driven Computing and Caching in 5G Networks: Architecture and Delay Analysis. *IEEE Wireless Communications* 25, 1 (February 2018), 70–75.

[6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 301–314.

[7] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. 2015. CYRUS: Towards Client-defined Cloud Storage. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 17, 16 pages.

[8] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz. 2015. Near optimal placement of virtual network functions. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 1346–1354.

[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 49–62.

[10] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus: Mapping Scientific Workflows onto the Grid. In *Grid Computing*, Marios D. Dikaiakos (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 11–20.

[11] Harishchandra Dubey, Jing Yang, Nick Constant, Amir Mohammad Amiri, Qing Yang, and Kunal Makodiya. 2015. Fog Data: Enhancing Telehealth Big Data Through Fog Computing. In *Proceedings of the ASE BigData & SocialInformatics 2015 (ASE BD&SI '15)*. ACM, New York, NY, USA, Article 14, 6 pages.

[12] H. Feng, J. Llorca, A. M. Tulino, and A. F. Molisch. 2018. Optimal Dynamic Cloud Network Control. *IEEE/ACM Transactions on Networking* (2018), 1–14.

[13] K. Kamran, E.M. Yeh, and Q. Ma. 2018. "Joint Computation, Forwarding and Data Placement in Data-Driven Computing Networks". https://www.dropbox.com/s/o2ytpk9mconn1vi/Optimal_Data_Driven_Computation_fullpaper.pdf?dl=0. (2018).

[14] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. 2012. Cuckoo: A Computation Offloading Framework for Smartphones. In *Mobile Computing, Applications, and Services*, Martin Gris and Guang Yang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 59–79.

[15] Aleksandra Knezevic, Quynh Nguyen, Jason A. Tran, Pradipta Ghosh, Pranav Sakulkar, Bhaskar Krishnamachari, and Murali Annavaram. 2017. CIRCE - a Runtime Scheduler for DAG-based Dispersed Computing: Demo. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)*. ACM, New York, NY, USA, Article 31, 2 pages.

[16] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. 2012. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE INFOCOM*. 945–953.

[17] C. Long, Y. Cao, T. Jiang, and Q. Zhang. 2018. Edge Computing Framework for Cooperative Video Processing in Multimedia IoT Systems. *IEEE Transactions on Multimedia* 20, 5 (May 2018), 1126–1139.

[18] Aleksandra Knezevic Jiatong Wang Quynh Nguyen Jason Tran H.V. Krishna Giri Narra Zhifeng Lin Songze Li Ming Yu Bhaskar Krishnamachari Salman Avestimehr Murali Annavaram Pranav Sakulkar, Pradipta Ghosh. 2018. WAVE: A Distributed Scheduling Framework for Dispersed Computing. *USC ANRG Technical Report* (2018).

[19] S. Sarkar, S. Chatterjee, and S. Misra. 2018. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Transactions on Cloud Computing* 6, 1 (Jan 2018), 46–59.

[20] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (Jan 2017), 30–39.

[21] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (March 2002), 260–274.

[22] Y. Wen, W. Zhang, and H. Luo. 2012. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. In *2012 Proceedings IEEE INFOCOM*. 2716–2720.

[23] Chien-Sheng Yang, Ramtin Pedarsani, and Amir Salman Avestimehr. 2018. Communication-Aware Scheduling of Serial Tasks for Dispersed Computing. *CoRR* abs/1804.06468 (2018). arXiv:1804.06468

[24] L. Yang, J. Cao, S. Tang, T. Li, and A. T. S. Chan. 2012. A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing. In *2012 IEEE Fifth International Conference on Cloud Computing*. 794–802.

[25] Edmund Yeh, Tracey Ho, Ying Cui, Michael Burd, Ran Liu, and Derek Leong. 2014. VIP: A Framework for Joint Dynamic Forwarding and Caching in Named Data Networks. In *Proceedings of the 1st ACM Conference on Information-Centric Networking (ACM-ICN '14)*. ACM, New York, NY, USA, 117–126.

[26] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu. 2016. Joint Optimization of Task Scheduling and Image Placement in Fog Computing Supported Software-Defined Embedded System. *IEEE Trans. Comput.* 65, 12 (Dec 2016), 3702–3712.

[27] J. Zhang, A. Sinha, J. Llorca, A. Tulino, and E. Modiano. 2018. Optimal Control of Distributed Computing Networks with Mixed-Cast Traffic Flows. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 1880–1888.

[28] J. Zhang, F. Wang, K. Wang, W. Lin, X. Xu, and C. Chen. 2011. Data-Driven Intelligent Transportation Systems: A Survey. *IEEE Transactions on Intelligent Transportation Systems* 12, 4 (Dec 2011), 1624–1639.