# Generic and Automatic Address Configuration for Data Center Networks[*]

Kai Chen[⋆†], Chuanxiong Guo[†], Haitao Wu[†], Jing Yuan[‡⋆], Zhenqian Feng[♯†],
Yan Chen[⋆], Songwu Lu[§], Wenfei Wu[♮†]
[⋆]Northwestern University, [†]Microsoft Research Asia, [‡]Tsinghua University, [♯]NUDT, [§]UCLA, [♮]BUAA
[⋆]{kchen,ychen}@northwestern.edu, [†]{chguo,hwu,v-zhfe,v-wenfwu}@microsoft.com,
[‡]yuan-j05@mails.tsinghua.edu.cn, [§]slu@cs.ucla.edu

## ABSTRACT

Data center networks encode *locality* and *topology* information into their server and switch addresses for performance and routing purposes. For this reason, the traditional address configuration protocols such as DHCP require huge amount of manual input, leaving them error-prone.

In this paper, we present DAC, a generic and automatic Data center Address Configuration system. With an automatically generated blueprint which defines the connections of servers and switches labeled by *logical IDs*, *e.g.*, IP addresses, DAC first learns the physical topology labeled by *device IDs*, *e.g.*, MAC addresses. Then at the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC makes an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem, and solves it with low time-complexity by leveraging the attributes of data center network topologies. Its malfunction detection scheme detects errors such as device and link failures and miswirings, including the most difficult case where miswirings do not cause any node degree change.

We have evaluated DAC via simulation, implementation and experiments. Our simulation results show that DAC can accurately find all the hardest-to-detect malfunctions and can autoconfigure a large data center with 3.8 million devices in 46 seconds. In our implementation, we successfully autoconfigure a small 64-server BCube network within 300 milliseconds and show that DAC is a viable solution for data center autoconfiguration.

## Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.3 [**Network Operations**]: Network management

## General Terms

Algorithms, Design, Performance, Management

## Keywords

Data center networks, Address configuration, Graph isomorphism

[*]This work was performed when Kai, Zhenqian and Wenfei were interns at Microsoft Research Asia.

## 1. INTRODUCTION

### 1.1 Motivation

Mega data centers [1] are being built around the world to provide various cloud computing services such as Web search, online social networking, online office and IT infrastructure out-sourcing for both individual users and organizations. To take the advantage of economies of scale, it is common for a data center to contain tens or even hundreds of thousands of servers. The current choice for building data centers is using commodity servers and Ethernet switches for hardware and the standard TCP/IP protocol suite for inter-server communication. This choice provides the best performance to price trade-off [2]. All the servers are connected via network switches to form a large distributed system.

Before the servers and switches can provide any useful services, however, they must be correctly configured. For existing data centers using the TCP/IP protocol, the configuration includes assigning IP address to every server. For layer-2 Ethernet, we can use DHCP [3] for dynamic IP address configuration. But servers in a data center need more than one IP address in certain address ranges. This is because for performance and fault tolerance reasons, servers need to know the locality of other servers. For example, in a distributed file system [4], a chunk of data is replicated several times, typically three, to increase reliability. It is better to put the second replica on a server in the same rack as the original, and the third replica on a server at another rack. The current practice is to embed locality information into IP addresses. The address locality can also be used to increase performance. For example, instead of fetching a piece of data from a distant server, we can retrieve the same piece of data from a closer one. This kind of locality based optimization is widely used in data center applications [4, 5].

The newly proposed data center network (DCN) structures [6–9] go one step further by encoding their topology information into their logical IDs. These logical IDs can take the form of IP address (e.g., in VL2 [9]), or MAC address (e.g., in Portland [8]), or even newly invented IDs (e.g., in DCell [6] and BCube [7]). These structures then leverage the topological information embedded in the logical IDs for scalable and efficient routing. For example, Portland switches choose a routing path by exploiting the location information of destination PMAC. BCube servers build a source routing path by modifying one digit at one step based on source and destination BCube IDs.

For all the cases above, we need to configure the logical IDs, which may be IP or MAC addresses or BCube or DCell IDs, for all the servers and switches. Meanwhile, in the physical topology, all the devices are identified by their unique device IDs, such as MAC addresses. A naïve way is to build a static device-to-logical ID mapping table at the DHCP server. Building such a table is mainly a manual effort which does not work for the following two reasons. First of all, the scale of data center is huge. It is not uncommon that a mega data center can have hundreds of thousands of servers [1]. Secondly, manual configuration is error-prone. A recent survey from 100 data center professionals [10] suggested that 57% of the data center outages are caused by human errors. Two more surveys [11, 12] showed 50%-80% of network downtime is due to human configuration errors. In short, "the vast majority of failures in data centers are caused, triggered or exacerbated by human errors." [13]

## 1.2 Challenges and Contributions

Automatic address configuration is therefore highly desirable for data center networks. We envision that a good autoconfiguration system will have the following features which also pose challenges for building such a system.

- Generality. The system needs to be applicable to various network topologies and addressing schemes.
- Efficiency and scalability. The system should assign a logical ID to a device quickly and be scalable to a large number of devices.
- Malfunction and error handling. The system must be able to handle various malfunctions such as broken NICs and wires, and human errors such as miswirings.
- Minimal human intervention. The system should require minimal manual effort to reduce human errors.

To the best of our knowledge, there are very few existing solutions and none of them can meet all the requirements above. In this paper, we address these problems by proposing DAC – a generic and automatic Data center Address Configuration system for *all* the existing and future data center networks. To make our solution generic, we assume that we only have a blueprint of the to-be-configured data center network, which defines how the servers and switches are connected, and labels each device with a logical ID. The blueprint can be automatically generated because all the existing data center network structures are quite regular and can be described either recursively or iteratively (see [6–9] for examples).

Through a physical network topology learning procedure, DAC first automatically learns and stores the physical topology of the data center network into an autoconfiguration manager. Then, we make the following two key contributions when designing DAC.

First of all, we solve the core problem of autoconfiguration: how to map the device IDs in the physical topology to the logical IDs in the blueprint while preserving the topological relationship of these devices? DAC makes an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism (GI) problem [14] in graph theory. Existing GI solutions are too slow for some large scale data center networks. Based on the attributes of data center network topologies, such as sparsity and symmetry (or asymmetry), we apply graph theory knowledge to design an optimization algorithm which significantly speed up the mapping. Specifically, we use three heuristics: *candidate selection via SPLD*, *candidate pruning via orbit* and *selective splitting*. The first heuristic is our own. The last two we selected from previous work [15] and [16] respectively, after finding that they are quite effective for data center graphs.

Secondly, despite that the malfunction detection problem is NP-complete and APX-hard[1], we design a practical scheme that subtly exploits the degree regularity in all data center structures to detect the malfunctions causing device degree change. For the hardest one with no degree change, we propose a scheme to compare the blueprint graph and the physical topology graph from multiple anchor points and correlate malfunctions via majority voting. Evaluation shows that our solution is fast and is able to detect all the hardest-to-detect malfunctions.

[1]A problem is APX-hard if there is no polynomial-time approximation scheme.

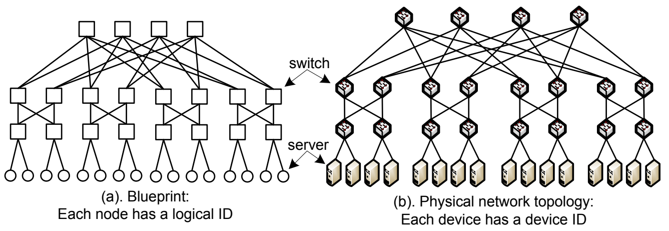
switch
server
(a). Blueprint:
Each node has a logical ID
(b). Physical network topology:
Each device has a device ID

**Figure 1: An example of blueprint, and physical topology constructed by following the interconnections in blueprint.**

We have studied our DAC design via extensive experiments and simulations. The experimental results show that the time of our device-to-logical ID mapping scales in proportion to the total number of devices in the networks. Furthermore, our simulation results show that DAC can autoconfigure a large data center with 3.8 million devices in 46 seconds. We have also developed and implemented DAC as an application on a 64-server testbed, where the 64 servers and 16 mini-switches form a two level BCube [7] network. Our autoconfiguration protocols automatically and accurately assign BCube logical IDs to these 64 servers within 300 milliseconds.

The rest of the paper is organized as follows. Section 2 presents the system overview. Section 3 introduces the device-to-logical ID mapping. Section 4 discusses how DAC deals with malfunctions. Section 5 and Section 6 evaluate DAC via experiments, simulations and implementations. Section 7 discusses the related work. Section 8 concludes the paper.

## 2. SYSTEM OVERVIEW

One important characteristic shared by all data centers is that a given data center is owned and operated by a single organization. DAC takes advantage of this property to employ a centralized autoconfiguration manager, which we call *DAC manager* throughout this paper. DAC manager deals with all the address configuration intelligences such as physical topology collection, device-to-logical ID mapping, logical ID dissemination and malfunction detection. In our design, DAC manager can simply be a server in the physical topology or can run on a separate control network.

Our centralized design is also inspired by the success of several, recent large-scale infrastructure deployments. For instance, the data processing system MapReduce [5] and the modern storage GFS [4] employ a central master at the scale of tens of thousands of devices. More recently, Portland [8] leverages a fabric manager to realize a scalable and efficient layer-2 data center network fabric.

As stated in our first design goal, DAC should be a generic solution for various topologies and addressing schemes. To achieve this, DAC cannot assume any specific form of structure or addressing scheme in its design. Considering this, DAC only uses the following two graphs as its input:

**1. Blueprint.** Data centers have well-defined structures. Prior to deploying a real data center, a blueprint (Figure 1a) should be designed to guide the construction of the data center. To make our solution generic, we only require the blueprint to provide the following minimal information:

- **Interconnections between devices.** It should define the interconnections between devices. Note that though it is possible for a blueprint to label port numbers and define how the ports of neighboring devices are connected, DAC does not depend on such information. DAC only requires the neighbor information of the devices, contained in any connected graph.

- **Logical ID for each device.** It should specify a logical ID for each device[2]. The encoding of these logical IDs conveys the topological information of the network structure. These logical IDs are vital for server communication and routing protocols.

Since data center networks are quite regular and can be described iteratively or recursively, we can automatically generate the blueprint using software.

**2. Physical network topology.** The physical topology (Figure 1b) is constructed by following the interconnections defined in the blueprint. In this physical topology, we use the MAC address as a device ID to uniquely identify a device. For a device with multiple MAC addresses, we use the lowest one.

In the rest of the paper, we use $G_b = (V_b, E_b)$ to denote the blueprint graph and $G_p = (V_p, E_p)$ to denote the physical topology graph. $V_b/V_p$ are the set of nodes (*i.e.*, devices) with logical/device IDs respectively, and $E_b/E_p$ are the set of edges (*i.e.*, links). Note that while the blueprint graph $G_b$ is known for any data center, the physical topology graph $G_p$ is not known until the data center is built and information collected.

The whole DAC system structure is illustrated in Figure 2. The two core components of DAC are *device-to-logical ID mapping* and *malfunction detection and handling*. We also have a module to collect the physical topology, and a module to disseminate the logical IDs to individual devices after DAC manager finishes the device-to-logical ID mapping. In what follows, we overview the design of these modules.

**1. Physical topology collection.** In order to perform logical ID resolution, we need to know both blueprint $G_b$ and physical topology $G_p$. Since $G_p$ is not known readily, DAC requires a communication channel over the physical network to collect the physical topology information. To this end, we propose a Communication channel Building Protocol (CBP). The channel built from CBP is a layered spanning tree and the root is DAC manager with level 0, its children are level 1, so on and so forth.

When the channel is built, the next step is to collect the physical topology $G_p$. For this, we introduce a Physical topology Collection Protocol (PCP). In PCP, the physical topology information, *i.e.*, the connection information between each node, is propagated bottom-up from the leaf devices to the root (*i.e.*, DAC manager) layer by layer. After $G_p$ is collected by DAC manager, we go to the device-to-logical ID mapping module.

**2. Device-to-logical ID mapping.** After $G_p$ has been collected, we come to device-to-logical ID mapping, which is a key component of DAC. As introduced in Section 1, the challenge is how to have the mapping reflect the topological relationship of these devices. To this end, we devise $O_2$, a fast one-to-one mapping engine, to realize this functionality. We elaborate this fully in Section 3.

**3. Logical ID dissemination.** When logical IDs for all the devices have been resolved, *i.e.*, the device-to-logical ID mapping table is achieved, we need to disseminate this information to the whole network. To this end, we introduce a Logical ID Dissemination Protocol (LDP). In contrast to PCP, in LDP the mapping table is delivered top-down from DAC manager to the leaf devices, layer by layer. Upon receipt of such information, a device can easily index its logical ID according to its device ID. A more detailed explanation of LDP together with CBP and PCP above is introduced in Section 5.

**4. Malfunction detection and handling.** DAC needs to automatically detect malfunctions and pinpoint their locations. For this, we introduce a malfunction detection and handling module. In DAC, this module interacts tightly with the *device-to-logical ID mapping* module because the former one is only triggered by the latter. If there exist malfunctions in $G_p$, our $O_2$ engine quickly perceives this by noticing that the physical topology graph $G_p$ mismatches with the blueprint graph $G_b$. Then, the malfunction detection module is immediately invoked to detect those malfunctioning devices and report them to network administrators. We describe this module in Section 4.

[2]While most data center structures, like BCube [7], DCell [6], Ficonn [17] and Portland [8], use device based logical ID, there also exist structures, like VL2 [9], that use port based logical ID. For brevity, in this paper, DAC is introduced and evaluated as the device based case. It can handle the port based scenario by simply considering each port as a single device and treating a device with multiple ports as multiple logical devices.

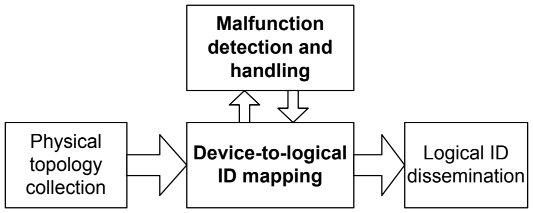
Malfunction detection and handling
Physical topology collection
Device-to-logical ID mapping
Logical ID dissemination

**Figure 2: The DAC system framework with four modules.**

## 3. DEVICE-TO-LOGICAL ID MAPPING

In this section, we formally introduce how DAC performs the device-to-logical ID mapping. We first formulate the mapping using graph theory. Then, we solve the problem via optimizations designed for data center structures. Last, we discuss how to do the mapping for data center expansion.

### 3.1 Problem Formulation and Solution Overview

As introduced, the challenge here is to do the device-to-logical mapping such that this mapping reflects the topological relationship of these devices. Considering we have the blueprint graph $G_b = (V_b, E_b)$ and the physical topology graph $G_p = (V_p, E_p)$, to meet the above requirement, we formulate the mapping problem as finding a one-to-one mapping between nodes in $V_p$ and $V_b$ while preserving the adjacencies in $E_p$ and $E_b$. Interestingly, this is actually a variant of the classical graph isomorphism (GI) problem [14].

DEFINITION 1. *Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, denoted by $G_1 \cong G_2$, if there is a bijection $f : V_1 \to V_2$ such that $\{u, v\} \in E_1$ if, and only if, $\{f(u), f(v)\} \in E_2$, for all $u, v \in V_1$. Such a bijection $f$ is called a graph isomorphism between $G_1$ and $G_2$.*

To the best of our knowledge, we are the first one to introduce the GI model to data center networks, thus solving the address autoconfiguration problem. After the problem formulation, the next step is to solve the GI problem. In the past 20 years, many research efforts have been made to determine whether the general GI problem is in P or NP [14]. When the maximum node degree is bounded, polynomial algorithm with $n^{O(d^2)}$ time complexity is known [18], where $n$ is the number of nodes and $d$ is the maximal node degree.

However, $n^{O(d^2)}$ is too slow for our problem since data centers can have millions of devices [6] and the maximal node degree can be more than 100 [9]. To this end, we devise $O_2$, a fast one-to-one mapping engine. As shown in Figure 3, $O_2$ starts with a base algorithm (*i.e.*, *$O_2$_Base_Mapping()*) for general graphs, and upon that we propose an optimization algorithm (*i.e.*, *$O_2$_Mapping()*) using three heuristics: *candidate selection via SPLD*, *candidate filtering via orbit* and *selective splitting* that are specially tailored for the attributes of data center structures and our real address autoconfiguration application. In the following, we first introduce some preliminaries together with the base algorithm, and then introduce the optimization algorithm.

### 3.2 The Base Algorithm

**Preliminaries.** Given a graph $G = (V, E)$, a **partition** of a vertex set $V$, *e.g.*, $\Pi = (\pi^0, \pi^1, \cdots, \pi^{n-1})$, is a set of disjoint non-empty

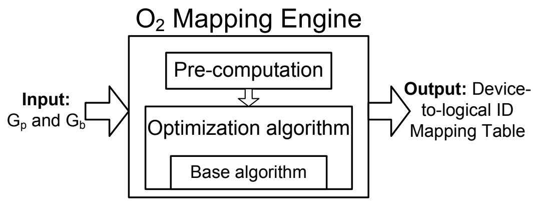

**Figure 3: The O$_2$ mapping engine.**

subsets of $V$ whose union is $V$. We call each subset $\pi^i(0 \leq i \leq n-1)$ a **cell**. In O$_2$, the basic operations on partitions or cells are "decompose" and "split".

- **Decompose.** Given a node $v$, a cell $\pi^i$ and a partition $\Pi$ where $v \in \pi^i$ and $\pi^i \in \Pi$, using $v$ to decompose $\pi^i$ means to replace $\pi^i$ with $\{v\}$ and $\pi^i \setminus v$ in partition $\Pi$, where $\setminus$ is set minus meaning to remove node $v$ from $\pi^i$.
- **Split.** Given two cells $\pi^i, \pi^t \in \Pi$, using $\pi^i$ to split $\pi^t$ means to do the following: first, for each node $v \in \pi^t$, we calculate a value $k = \eta(v, \pi^i)$ as the number of connections between node $v$ and nodes in $\pi^i$ where $\eta$ is called connection function; then, we divide $\pi^t$ into smaller cells by grouping the nodes with the same $k$ value together to be a new cell. Moreover, we call $\pi^i$ the inducing cell and $\pi^t$ the target cell. The target cell should be a non-singleton.

A partition is **equitable** if no cell can be split by any other cell in the partition. A partition is **discrete** if each cell of this partition is a singleton (*i.e.*, single element). Suppose we use an inducing cell pair $\pi_1^i/\pi_2^i$ to split target cell pair $\pi_1^t/\pi_2^t$ respectively, $\pi_1^t/\pi_2^t$ are **divided isomorphically** by $\pi_1^i/\pi_2^i$ if for each value $k = 0, 1, 2, \cdots$, $\pi_1^t$ has the same number of nodes with $k$-connection to $\pi_1^i$ as $\pi_2^t$ has to $\pi_2^i$.

Note that the cells in a partition have their orders. We use parenthesis to represent a partition, and each cell is indexed by its order. For example, $\Pi = (\pi^0, \pi^1, \cdots, \pi^{n-1})$ means a partition $\Pi$ with $n$ cells and the $i$-th cell is $\pi^{i-1}$. In our mapping algorithm, decomposition/split operation always works on the corresponding pair of cells (*i.e.*, two cells with the same order) in two partitions. Furthermore, during these operations, we place the split cells back to the partitions in corresponding orders. For example, decomposing $\pi_1^i/\pi_2^i$ with $v/v'$, we replace $\pi_1^i$ with $\{v\}, \pi_1^i \setminus v$ and $\pi_2^i$ with $\{v'\}, \pi_2^i \setminus v'$, and then place the split cells back to the partitions such that $\{v\}$ and $\{v'\}$ are in the same order, $\pi_1^i \setminus v$ and $\pi_2^i \setminus v'$ are in the same order.

In addition to the above terms, we further have two important terms used in the optimization algorithm, which are **SPLD** and **orbit**.

- **SPLD.** SPLD is short for Shortest Path Length Distribution, the SPLD of a node $v$ is the distribution of the distances between this node and all the other nodes in the graph.
- **Orbit.** An orbit is a subset of nodes in graph $G$ such that two nodes $u$ and $v$ are in the same orbit if there exist an automorphism[3] of $G$ that maps $u$ to $v$ [19]. For example, in $G_b$ of Figure 6, $l_1$ to $l_2$ are in the same orbit since there is an automorphism permutation of $G_b$, which is ($l_2$ $l_1$ $l_3$ $l_4$ $l_5$ $l_6$ $l_7$ $l_8$), that maps $l_1$ to $l_2$.

**Base algorithm.** Figure 4 is a base mapping algorithm for general graphs we summarize from previous literatures. It contains *Decomposition()* and *Refinement()*, and it repeatedly decomposes and refines (or splits) $\Pi_p$ and $\Pi_b$ until either they both are discrete, or it terminates in the middle finding that $G_p$ and $G_b$ are not isomorphic.

In each level of recursion, we first check if the current partitions $\Pi_p$ and $\Pi_b$ are discrete. If so, we return $true$ (line 2) and get a one-to-one mapping by mapping each singleton cell of $\Pi_p$ to the corresponding singleton cell of $\Pi_b$. Otherwise, we do *Decomposition()*.

[3]An automorphism of a graph is a graph isomorphism with itself, i.e., a mapping from the vertices of the given graph $G$ back to vertices of $G$ such that the resulting graph is isomorphic with $G$.

```
O2_Base_Mapping($\Pi_p, \Pi_b$) /* Initially, $\Pi_p = (V_p)$ and $\Pi_b = (V_b)$. */
1   if ($\Pi_p$ and $\Pi_b$ are both discrete)
2       return $true$;
3   else
4       select a vertex $v \in \pi_p^i$; /* $\pi_p^i$ is nonsingleton. */
5       foreach vertex $v' \in \pi_b^i$
6           ($\Pi_p, \Pi_b$) = Decomposition($\Pi_p, \pi_p^i, v, \Pi_b, \pi_b^i, v'$);
7           if (Refinement($\Pi_p, \Pi_b$) == $true$)
8               if (O2_Base_Mapping($\Pi_p, \Pi_b$) == $true$)
9                   return $true$;
10              else continue;
11          else continue;
12      return $false$;
```

**Figure 4: Pseudocode of the generic algorithm for one-to-one mapping (*i.e.*, graph isomorphism). For clarity, the functions *Decomposition()* and *Refinement()* are explained in the context.**

In *Decomposition()*, we first select a pair of corresponding nonsingleton cells $\pi_p^i$ and $\pi_b^i$, and then select a pair of nodes $v \in \pi_p^i$ and $v' \in \pi_b^i$ to decompose $\pi_p^i$ and $\pi_b^i$ respectively (lines 4-6). The partitions $\Pi_p$ and $\Pi_b$ then become more concrete: $\Pi_p = (\pi_p^0, \cdots, \{v\}, \pi_p^i \setminus v, \cdots, \pi_p^k)$ and $\Pi_b = (\pi_b^0, \cdots, \{v'\}, \pi_b^i \setminus v', \cdots, \pi_b^k)$. Immediately after decomposition, we do *Refinement()* on $\Pi_p$ and $\Pi_b$ (line 7).

In *Refinement()*, we repeatedly try to use every newly born pair of cells to split *all* other corresponding nonsingleton pairs of cells. For each pair of cells that have been simultaneously divided, we check whether the two cells are divided isomorphically or not. If not, then *Refinement*($\Pi_p$, $\Pi_b$) returns *false*. Otherwise, if each time the pair of target cells are isomorphically divided, *Refinement*($\Pi_p$, $\Pi_b$) will continue until $\Pi_p$ and $\Pi_b$ are equitable and returns *true*.

If *Refinement*($\Pi_p$, $\Pi_b$) returns *true*, we go one step further of recursion to work on new equitable partitions (line 8). Otherwise, it means that $v'$ cannot be mapped to $v$ and we try the next candidate in $\pi_b^i$ (line 11). If all the candidates in $\pi_b^i$ fail to be mapped to $v$, we must backtrack (line 10). Such recursion continues until either both partitions become discrete, *i.e.*, a one-to-one mapping is found (line 2), or we backtrack to root of the search tree, thus concluding that no one-to-one mapping exists (line 12).

## 3.3 The Optimization Algorithm

Compared with general graphs, network topologies of data centers have the following attributes: 1) They are sparse; 2) They are typically either highly symmetric like BCube [7] or highly asymmetric like DCell [6]. In any case, for our address autoconfiguration problem, the blueprint graph is available in advance which means we can do some precomputation.

Based on these features, we apply graph theory to design an optimization algorithm with three heuristics: *candidate selection via SPLD*, *candidate filtering via orbit* and *selective splitting* to speedup the device-to-logical ID mapping. Specifically, we introduce the first heuristic, and borrow the last two from [15] and [16] respectively, based on their effectiveness for graphs derived for data centers. Our experiments in Section 6.2 indicate that we need all these three heuristics to solve our problem and any partial combination of them is slow for some structures. Figure 5 is the optimization algorithm built on the base algorithm. In the following, we explain the three heuristics emphasizing the reasons why they are suitable for data center graphs.

**1. Candidate selection via SPLD.** We observe that nodes in data centers have different roles such as switches and servers, and switches in some data centers like FatTree can be further divided into ToR, aggregation and core. Hence from this point of view, SPLD can be helpful by itself to distinguish nodes of different roles. Furthermore, SPLD can provide even significant improvement for structures like DCell which are very asymmetric. This is because the SPLDs of different nodes in DCell are very different. To take advantage of this

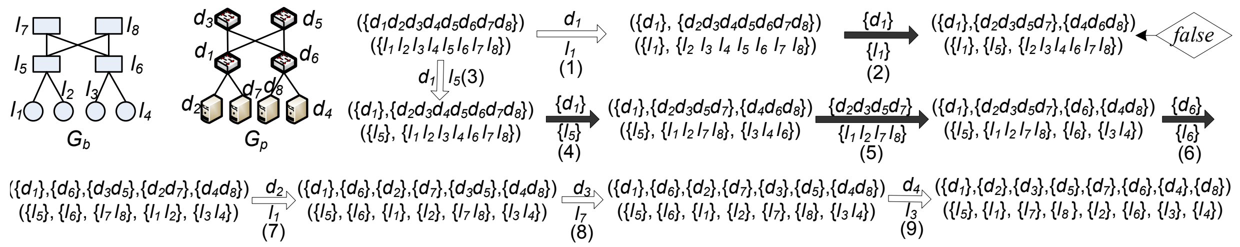
l7
l8
l5
l6
l1
l2
l3
l4
Gb
d3
d5
d1
d6
d2
d7
d8
d4
Gp
({d1d2d3d4d5d6d7d8})
({l1 l2 l3 l4 l5 l6 l7 l8})
d1
l1
(1)
({d1}, {d2d3d4d5d6d7d8})
({l1}, {l2 l3 l4 l5 l6 l7 l8})
{d1}
{l1}
(2)
({d1},{d2d3d5d7},{d4d6d8})
({l1}, {l5}, {l2 l3 l4 l6 l7 l8})
false
d1 l5(3)
({d1},{d2d3d4d5d6d7d8})
({l5}, {l1 l2 l3 l4 l6 l7 l8})
{d1}
{l5}
(4)
({d1},{d2d3d5d7},{d4d6d8})
({l5}, {l1 l2 l7 l8}, {l3 l4 l6})
{d2d3d5d7}
{l1 l2 l7 l8}
(5)
({d1},{d2d3d5d7},{d6},{d4d8})
({l5}, {l1 l2 l7 l8}, {l6}, {l3 l4})
{d6}
{l6}
(6)
({d1},{d6},{d3d5},{d2d7},{d4d8})
({l5}, {l6}, {l7 l8}, {l1 l2}, {l3 l4})
d2
l1
(7)
({d1},{d6},{d2},{d7},{d3d5},{d4d8})
({l5}, {l6}, {l1}, {l2}, {l7 l8}, {l3 l4})
d3
l7
(8)
({d1},{d6},{d2},{d7},{d3},{d5},{d4d8})
({l5}, {l6}, {l1}, {l2}, {l7}, {l8}, {l3 l4})
d4
l3
(9)
({d1},{d2},{d3},{d5},{d7},{d6},{d4},{d8})
({l5}, {l1}, {l7}, {l8}, {l2}, {l6}, {l3}, {l4})

**Figure 6: An example of mapping between $G_p$ and $G_b$. White arrow is decomposition and dark arrow is refinement.**

```
/* pre-compute the SPLDs for all nodes in $G_b$; */
/* pre-compute all the orbits in $G_b$; */
O2_Mapping($\Pi_p, \Pi_b$) /* Initially, $\Pi_p = (V_p)$ and $\Pi_b = (V_b)$. */
1   if ($\Pi_p$ and $\Pi_b$ are both discrete)
2       return $true$;
3   else
4       select a vertex $v \in \pi_p^i$; /* $\pi_p^i$ is nonsingleton. */
5           let the candidate node pool $cp = \pi_b^i$;
6           if ($|cp| > th1$ && $|\text{SPLD}(cp)| > th2$) /* thresholds */
7               compute SPLD($v$) and then delete all nodes from $cp$
                having different SPLDs from SPLD($v$);
8           select a vertex $v' \in cp$;
9           ($\Pi_p, \Pi_b$) = Decomposition($\Pi_p, \pi_p^i, v, \Pi_b, \pi_b^i, v'$);
10          bool refineSucceed = $true$;
11          if (Refinement*($\Pi_p, \Pi_b$) == $true$)
12              if (O2_Mapping($\Pi_p, \Pi_b$) == $true$)
13                  return $true$;
14              else refineSucceed = $false$;
15          else refineSucceed = $false$;
16          delete $v'$ and all its equivalent nodes from $cp$;
17          if (!refineSucceed && !empty($cp$))
18              goto line 8;
19      return $false$;
```

**Figure 5: Pseudocode of the optimization algorithm for data center graphs. For clarity, *Refinement**()* is explained in the context.**

property, we propose using SPLD as a more sophisticated signature to select mapping candidates. That is, when we try to select a node $v'$ in $G_b$ as a candidate to be mapped to a node $v$ in $G_p$, we only select the $v'$ from these nodes that have the same SPLD as $v$. This is effective because two nodes with different SPLDs cannot be mapped to each other. However, computing SPLDs for all nodes in a large graph requires time. Fortunately, in our case, this can be computed earlier on the blueprint.

In our optimization algorithm, we precompute the SPLDs for all nodes of $G_b$ beforehand. In lines 6-7, we improve the base algorithm in this way: if we find the number of candidates (*i.e.*, nodes in $cp$) for a node, say $v$ in $G_p$, to be mapped to is larger than a threshold $th1$ (*i.e.*, $|cp| > th1$) and the number of different SPLDs of them is larger than a threshold $th2$ (*i.e.*, $|\text{SPLD}(cp)| > th2$), we compute the SPLD for $v$ and only select candidates in $cp$ having the same SPLD. Thresholds $th1$ and $th2$ are tuneable. Note that using this heuristic is a tradeoff: although we can do precomputation on $G_b$ offline, applying this optimization means that we should compute SPLD($v$) online, which also consumes time. In all our experiments later, we apply this heuristic on all the structures only once at the first round of mapping.

**2. Candidate filtering via orbit.** It is indicated in [15] that for $v \in G$ and $v' \in G'$, if $v'$ cannot be mapped to $v$, all nodes in the same orbit as $v'$ cannot be mapped to $v$ either. We find this theory is naturally suited for solving the GI problem on data centers: First, some structures such as BCube are highly symmetric, and there should be many symmetric nodes within these structures that are in the same orbit. Second, the blueprint graph is available much earlier than the real address autoconfiguration stage, and we can easily precompute the orbits in the blueprint beforehand using preexisting tools such as [16, 20].

In Figure 4, the base algorithm tries to map $v$ to every node in $\pi_b^i$ iteratively if the current mapping fails which is not effective especially for highly symmetric data center structures. Observing this, in the optimization algorithm, we precompute all the orbits of $G_b$ beforehand. Then, as shown in lines 16-18, we improve the base algorithm: if we find a certain node $v'$ cannot be mapped to $v$, we skip all the attempts that try to map $v$ to any other node in the same orbit as $v'$, because according to above theory these nodes cannot be mapped to $v$ either.

**3. Selective splitting.** In the base algorithm, *Refinement()* tries to use the inducing cell to split all the other cells. As data center structures are sparse, it is likely that while there are many cells in the partition, the majority of them are disjoint with the inducing cell. Observing this, in line 11, we use *Refinement**() in which we only try to split the cells that really connect to the inducing cell other than all[4].

Furthermore, when splitting a connected cell $\pi^t$, the base algorithm tries to calculate the number of connections between each node in $\pi^t$ and the inducing cell, and then divide $\pi^t$ based on these values. Again, due to sparsity, it is likely that the number of nodes in $\pi^t$ that really connect to the inducing cell is very small. Observing this, in a similar way, we speed up by only calculating the number of connections for the nodes actually connected. The unconnected nodes can be grouped together directly. Specifically, when splitting $\pi^t$ using inducing cell $\pi^i$, we first move the elements in $\pi^t$ with connections to $\pi^i$ to the left-end of $\pi^t$ and leave all unconnected elements on the right. Then, we only calculate the $k$ values for the elements on the left, and group them according to the values.

### *A Walkthrough Example for $O_2$.*

We provide a step by step example of our algorithm in Figure 6. $G_b$ is labeled by its logical IDs and $G_p$ is labeled by its device IDs. White arrows mean decomposition and dark arrows mean refinement. Suppose all orbits in $G_b$ have been calculated beforehand. In this case they are $\{\{l_1\ l_2\ l_3\ l_4\}, \{l_5\ l_6\}, \{l_7\ l_8\}\}$.

Initially, all nodes in $G_p/G_b$ are in one cell in partitions $\Pi_p/\Pi_b$. Step (1) decomposes original $\Pi_p/\Pi_b$ using $d_1/l_1$. Step (2) refines the current $\Pi_p/\Pi_b$ using inducing cells $\{d_1\}/\{l_1\}$, but fails due to a non-isomorphic division. This is because during splitting, $\{d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8\}$ has 4 elements with 1-connection to $\{d_1\}$ and 3 elements with 0-connection; while $\{l_2\ l_3\ l_4\ l_5\ l_6\ l_7\ l_8\}$ has 1 element with 1-connection to $\{l_1\}$ and 7 elements with 0-connection. Therefore, they are not divided isomorphically.

From step (2), we know $l_1$ cannot be mapped to $d_1$. By optimization heuristic 2, we skip the candidates $l_2, l_3$ and $l_4$ which are in the same orbit as $l_1$. So in Step (3), we decompose the original $\Pi_p/\Pi_b$ using $d_1/l_5$. Steps (4) refines the current $\Pi_p/\Pi_b$ using $\{d_1\}/\{l_1\}$. Specifically, in $\Pi_p$ we find $d_2, d_3, d_5$ and $d_7$ have 1-connection to $\{d_1\}$ while the rest do not, and in $\Pi_b$ we find $l_1, l_2, l_7$ and $l_8$ have

[4]We achieve this by maintaining an adjacency list which is built once when the graph is read. In the adjacency list, for each vertex, we keep the neighboring vertices, so at any point we know the vertices each vertex is connected to. We also have another data structure that keeps track of the place where each vertex is located at within the partition. In this way, we know which cell is connected to the inducing cell.

1-connection to $\{l_5\}$ while the rest do not. So $\Pi_p/\Pi_b$ are isomorphically divided by $\{d_1\}/\{l_1\}$. After step (4), since the current partitions $\Pi_p/\Pi_b$ are not yet equitable, in steps (5) and (6), we continuously use newly born cells $\{d_2\ d_3\ d_5\ d_7\}/\{l_1\ l_2\ l_7\ l_8\}$ and $\{d_6\}/\{l_6\}$ to further split other cells until $\Pi_p/\Pi_b$ are equitable.

Steps (7)-(9) decompose the current partitions using $d_2/l_1$, $d_3/l_7$ and $d_4/l_3$ respectively. Since in each of these 3 steps, there is no cell that can be split by other cells, no division is performed. After step (9), the two partitions $\Pi_p/\Pi_b$ are discrete and we find a one-to-one mapping between $G_p$ and $G_b$ by mapping each node in $\Pi_p$ to its corresponding node in $\Pi_b$.

Two things should be noted in the above example: First and most importantly, we do not use optimization heuristic 1 above since we want to show the case of non-isomorphic division in steps (1)-(2). In the real $O_2$ mapping, after applying heuristic 1, we will directly go from step (3) instead of trying to map $d_1$ to $l_1$ as above because they have different SPLDs. This shows that SPLD is effective in selecting mapping candidates. Second, although we have not explicitly mentioned optimization heuristic 3, in each refinement we only try to split the connected cells rather than all cells. For example, after step (7), $\{d_2\}/\{l_1\}$ are newly born, but when it comes to refinement, we do not try to split $\{d_3\ d_5\}/\{l_7\ l_8\}$ or $\{d_4\ d_8\}/\{l_3\ l_4\}$ using $\{d_2\}/\{l_1\}$ because they are disjoint.

## 3.4 Using $O_2$ for Data Center Expansion

To meet the growth of applications and storage, the scale of a data center does not remain the same for long [21]. Therefore, address autoconfiguration for data center expansion is required. Two direct approaches are either to configure the new part directly, or to configure the entire data center as a whole. However, both approaches have problems: the first one fails to take into account the connections between the new part and the old part of the expanded data center; the second one considers the connections between the new part and the old part, but it may cause another lethal problem, *i.e.*, the newly allocated logical IDs are different from the original ones for the same devices of the old part, messing up existing communications.

To avoid these problems, DAC configures the entire data center while keeping the logical IDs for the old part unmodified. To achieve this goal, we still use $O_2$ but need to modify the input. Instead of putting all the nodes from a graph in one cell as before, we first differentiate nodes between the new part and the old part in $G_p$ and $G_b$. Since we already have the device-to-logical ID mapping for the old part, say $d_i \rightarrow l_i$ for $0 \leq i \leq k$, we explicitly express such one-to-one mapping in the partitions. In other words, we have $\Pi_p = (\{d_0\}, \cdots, \{d_k\}, T_p)$ and $\Pi_b = (\{l_0\}, \cdots, \{l_k\}, T_b)$, all the nodes for the new part of $G_p/G_b$ are in $T_p/T_b$ respectively. Then, we refine $\Pi_p/\Pi_b$ until they both are equitable. At last, we enter $O_2$ mapping with the equitable partitions. In this way, we can produce a device-to-logical ID mapping table for the new part of data center while keeping the logical IDs for devices of the old part unmodified.

## 4. MALFUNCTION DETECTION AND HANDLING

As introduced before, the malfunction detection module is triggered when $O_2$ returns *false*. This "false" indicates the physical topology is not the same as the blueprint. In this section, we describe how DAC handles malfunctions.

## 4.1 Malfunction Overview

Malfunctions can be caused by hardware and software failures, or simply human configuration errors. For example, bad or mismatched network card and cables are common, and miswired or improperly connected cables are nearly inevitable.

| Structure | Degrees of switches | Degrees of servers |
|---|---|---|
| BCube$(n,k)$ | $n$ | $k+1$ |
| FatTree$(n)$ | $n$ | 1 |
| VL$(n_r,n_p)$ | $n_p,(n_r+2)$ | 1 |
| DCell$(n,k)$ | $n$ | $k+1$ |

**Table 1: Degree patterns in BCube, FatTree, VL2 and DCell structures. $n, k, n_r, n_p$ are the parameters to define these networks, they are fixed for a given structure.**

We consider and categorize three malfunction types in data centers: *node, link and miswiring*. The first type occurs when a given server or switch breaks down from hardware or software reasons, causing it to be completely unreachable and disconnected from the network; the second one occurs when the cable or network card is broken or not properly plugged in so that the connectivity between devices on that link is lost; the third one occurs when wired cables are different from those in the blueprint. These malfunctions may introduce severe problems and downgrade the performance.

Note that from the physical topology, it is unlikely to clearly distinguish some failure types, e.g., a crashed server versus completely malfunctioning interface cards on that server. *Our goal is to detect and further locate all malfunction-related devices, and report the device information to network administrators, rather than identifying the malfunction type*. We believe our malfunction handling not only solves this issue for autoconfiguration, but also reduces the deployment/maintenance costs for real-world large data center deployment.

## 4.2 Problem Complexity and Challenge

The problem of malfunction detection can be formally described as follows. Given $G_b$ and $G_p$, the problem to locate all the malfunctioning parts in the graph $G_p$ is equivalent to obtaining the maximum common subgraph (MCS) $G_{mcs}$ of $G_b$ and $G_p$. Thus, we compare $G_{mcs}$ with $G_p$ to find the differences, which are the malfunctioning parts. All the devices (*i.e.*, servers or switches) related to these parts, which we call *malfunctioning devices*, can be detected. However, it is proven that the MCS problem is NP-complete [22] and APX-hard [23]. That is, there is no efficient algorithm, especially for large graphs such as those of data center network topologies. Therefore, we resort to designing our own algorithms based on the particular properties of data center structures and our real-world application scenario. There are two problems we need to address in the following subsections: 1) detecting the malfunctioning devices by identifying their device IDs; 2) locating the physical position of a malfunctioning device with its device ID automatically.

## 4.3 Practical Malfunction Detection Methods

To achieve better performance and easier management, large-scale data centers are usually designed and constructed according to some patterns or rules. Such patterns or rules imply two properties of the data center structures: 1) the nodes in the topologies typically have regular degrees. For example, we show the degree patterns for several well-known data center networks in Table 1; 2) the graphs are sparse, so that our $O_2$ can quickly determine if two graphs are isomorphic. These properties are important for us to detect malfunctions in data centers. In DAC, the first property is used to detect malfunctioning devices where there are node degree changes, and the second one serves as a tool in our malfunction detection scheme for the case where no degree change occurs.

### *4.3.1 Malfunction with Node Degree Change*

For the aforementioned three types of malfunctions, we discuss them one by one as follows. Our observation is that most of the cases may cause the change of degree on devices.

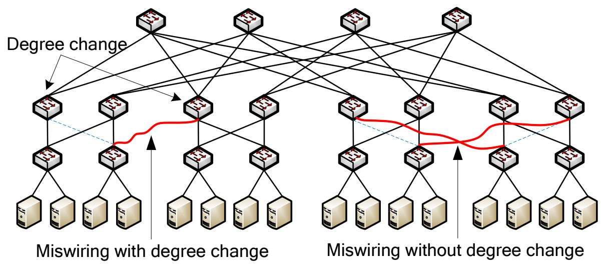

**Figure 7: Miswirings with and without degree change.**

- **Node.** If there is a malfunctioning node, the degrees of its neighboring nodes are decreased by one, and thus it is possible to identify the malfunction by checking its neighbor nodes.
- **Link.** If there is a malfunctioning link, the degrees of associated nodes are decreased by one, making it possible to detect.
- **Miswiring.** Miswirings are somewhat more complex than the other two errors. As shown in the left of Figure 7, the miswiring causes its related nodes to increase or decrease their degrees and can be detected readily. On the contrary, in the right of Figure 7, the miswirings of a pair of cables occur coincidentally so that the degree change caused by one miswired cable is glossed over by another, and thus no node degree change happens. We discuss this hardest case separately in the following.

Note that for any malfunction caused by the links, *i.e.*, link failure or miswirings, we report the associated nodes (*i.e.*, malfunctioning devices) in our malfunction detection.

### 4.3.2 Malfunction without Node Degree Change

Though in most cases the malfunctions cause detectable node degree change [24], it is still possible to have miswirings with no node degree change. This case occurs after an administrator has checked the network and the degree-changing malfunctions have been fixed. The practical assumptions here are: 1) the number of nodes involved in such malfunctions is a considerably small amount over all the nodes; 2) $G_p$ and $G_b$ have the same number of nodes as well as node degree patterns.

Despite the miswirings, the vast majority part of $G_p$ and $G_b$ are still the same. We leverage this fact to detect such miswirings. Our basic idea is that we first find some nodes that are supposed to be symmetric between $G_p$ and $G_b$, then use those nodes as anchor points to check if the subgraphs deduced from them are isomorphic. Through this we derive the difference between the two graphs, and correlate the malfunctioning candidates derived from different anchor points to make a decision. Basically, our scheme has two parts: anchor point selection and malfunction detection.

To minimize the human intervention, the first challenge is selecting anchor pairs between the blueprint graph $G_b$ and the physical topology graph $G_p$ without human input. Our idea is again to leverage the SPLD. Considering that the number of nodes involved in miswirings is small, it is likely that two "symmetric" nodes in two graphs will still have similar SPLDs. Based on this, we design our heuristics to select anchor pair points, which is *Anchor_Pair_Selection()* in Figure 8. In the algorithm, $||\text{SPLD}(v) - \text{SPLD}(v')||$ is simply the *Euclidean* distance. Given that two node with similar SPLDs are not necessarily a truly symmetric pair, our malfunction detection scheme will take the potential false positives into account, and handle this issue via majority voting.

Once the anchor node pairs have been selected, we compare $G_b$ and $G_p$ from these anchor node pairs and correlate malfunctions via majority voting. The algorithm for this is *Malfunction_Detection()* in

```
/* pre-compute the SPLDs of all nodes in $G_b$, select one node from each
   group of nodes with same SPLDs and store it in $C_b$ */
$(A_p, A_b) =$ Anchor_Pair_Selection$(G_p, G_b)$
1  $A_p =$ a group of selected anchor points in $G_p$;
2  foreach $v \in A_p$
3      select a $v' \in C_b$ that minimizes $||\text{SPLD}(v) - \text{SPLD}(v')||$;
4      store $v/v'$ in $A_p/A_b$;
Malfunction_Detection$(G_p, G_b, A_p, A_b)$
5  Define $S_p^x(v)$ as maximal subgraph of $G_p$ with maximal hop length $x$
   from node $v$ where $v \in G_p$, and the same as $S_b^x(v')$;
6  foreach pair of nodes $v/v' \in A_p/A_b$
7      use binary search to find a value $x$ that satisfies $O_2$_Mapping$(S_p^x(v),$
       $S_b^x(v')) = true$ and $O_2$_Mapping$(S_p^{x+1}(v), S_b^{x+1}(v')) = false$;
8      foreach node $i \in G_p$ that is $x$-hop or $(x+1)$-hop away from $v$
9          counter($i$)=counter($i$)+1;
10 return a node list sorted by their counter values;
```

**Figure 8: Pseudocode for malfunction detection.**

Figure 8. Specifically, given $G_p/G_b$, $A_p/A_b$ and definition of maximal subgraph $S_p^x(v)$ in line 5, for each anchor pair $v/v' \in A_p/A_b$, we search the maximal isomorphic subgraph of graphs $G_p/G_b$ with hop length $x$ from nodes $v/v'$ respectively. The process to obtain such subgraph is in line 7. We can use binary search to accelerate the searching procedure. If we find that $S_p^x(v)$ and $S_b^x(v')$ are isomorphic while $S_p^{x+1}(v)$ and $S_b^{x+1}(v')$ are not, we assume some miswirings happened between $x$-hop and $(x+1)$-hop away from $v$ and the nodes in these two hops are suspicious. In line 9, we increase a counter for each of these nodes to represent this conclusion.

After finishing the detection from all the anchor points, we report a list to the administrator. The list contains node device IDs and counter values of each node, ranked in the descending order of the counter values. Essentially, the larger its counter value, the more likely the device is miswired. Then the administrator will go through the list and rectify the miswirings. This process stops when he finds a node is not really miswired and ignores the rest of nodes on the list.

The accuracy of our scheme depends on the number of anchor points we selected for detection versus the number of miswirings in the network. Our experiments suggest that, with a sufficient number of anchor points, our algorithm can always find all the malfunctions (*i.e.*, put the miswired devices on top of the output list). According to the experimental results in Section 6.4, with at most 1.5% of nodes selected as anchor points we can detect all miswirings on the evaluated structures. To be more reliable, we can always conservatively select a larger percentage of anchor points to start our detection and most likely we will detect all miswirings (*i.e.*, have all of them on top of the list). Actually, this can be facilitated by the parallel computing because in our malfunction detection, the calculations from different anchor points are independent of each other and thus can be performed in parallel.

After fixing the miswirings, we will run $O_2$ to get the device-to-logical ID mapping again. Even in the case that not all the miswirings are on the top of the list and we miss some, $O_2$ will perceive that quickly. Then we will re-run our detection algorithm until all miswirings are detected and rectified, and $O_2$ can get the correct device-to-logical ID mapping finally.

## 4.4 Device Locating

Given a detected malfunctioning device, the next practical question is how to identify the location of the device given only its device ID (*i.e.*, MAC). In fact, the device locating procedure is not necessarily achieved by an autoconfiguration algorithm, but also possibly by some human efforts. In this paper, we argue that it is a practical deployment and maintenance problem in data centers, and thus we seek a scheme to collect such location information automatically.

Our idea is to sequentially turn on the power of each rack in order to generate a record for the location information. This procedure is performed only once and the generated record is used by the administrator to find a mapping between MAC and rack. It works as follows: 1) To power on the data center for the first time, the administrator turns on the power of server racks one by one sequentially. We require a time interval between powering each rack so we can differentiate devices in different racks. The time interval is a tradeoff: larger values allow easier rack differentiation while smaller values reduce boot time cost on all racks. We think by default it should be 10 seconds. 2) In the physical topology collection stage, when reporting the topology information to DAC manager, each device also piggybacks the boot-up time, from when it had been powered on to its first reporting. 3) When receiving such boot-up time information, DAC manager groups the devices with similar boot-up times (compared to the power on time interval between racks). 4) When DAC manager outputs a malfunctioning device, it also outputs the boot-up time for that group. Therefore, the administrator can check the rack physical position accordingly.

To summarize, our malfunction detection and locating designs focus on how to quickly detect and locate various malfunctions including the most difficult miswiring cases. We note that our schemes help to identify malfunctions, but not repair them. It is our hope that the detection procedure can help administrators to fix any malfunction more rapidly during the autoconfiguration stage.

## 5. IMPLEMENTATION AND EXPERIMENT

In this section, we first introduce the protocols that are used to do physical topology collection and logical ID dissemination. Then, we describe our implementation of DAC.

### 5.1 Communication Protocols

To achieve reliable physical topology collection and logical ID dissemination between all devices and DAC manager, we need a communication channel over the network. We note that the classical spanning tree protocol (STP) does not fit our scenario: 1) we have a fixed root - DAC manager, so network-wide broadcast for root selection is not necessary; 2) the scale of data center networks can be hundreds of thousands, making it difficult to guarantee reliability and information correctness in the network-wide broadcast. Therefore, we provide a Communication channel Building Protocol (CBP) to set up a communication channel over a mega-data center network. Moreover, we introduce two protocols, namely the Physical topology Collection Protocol (PCP) and the Logical ID Dissemination Protocol (LDP), to perform the topology information collection and ID dissemination over that spanning tree built by CBP.

**Building communication channel.** In CBP, each network device sends Channel Building Messages (CBMs) periodically (with a timeout interval $T$), to all of its interfaces. Neighbor nodes are discovered by receiving CBMs. Each node sends its own CBMs, and does not relay CBMs received from other nodes. To speed up the information propagation procedure, a node also sends out a CBM if it observes changes in neighbor information. A checking interval (*c-int*) is introduced to reduce the number of CBM messages by limiting the minimal interval between two successive CBMs.

DAC manager sends out its CBM with its level marked as 0, and its neighbor nodes correspondingly set their levels to 1. This procedure continues until all nodes get their respective levels, representing the number of hops from that node to DAC manager. A node randomly selects a neighbor node as its parent if that node has the lowest level among its neighbors, and claims itself as that node's child by its next CBM. The communication channel building procedure is finished once every node has its level and has selected its parent node. Therefore, the built communication channel is essentially a *layered* spanning tree, rooted at DAC manager. We define a *leaf node* as one

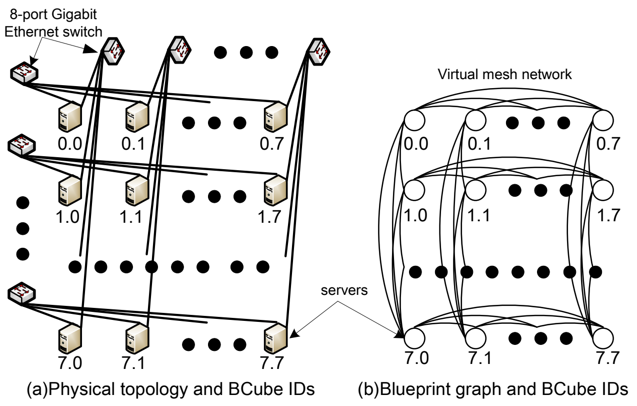
8-port Gigabit Ethernet switch
0.0
0.1
0.7
1.0
1.1
1.7
servers
7.0
7.1
7.7
(a)Physical topology and BCube IDs
Virtual mesh network
0.0
0.1
0.7
1.0
1.1
1.7
7.0
7.1
7.7
(b)Blueprint graph and BCube IDs

**Figure 9: The testbed topology and blueprint.**

that has the largest level among its neighbors and no children node. If a leaf node observes no neighbor updates for a timeout value, $3 * T$, it enters the next stage, physical topology information collection.

**Physical topology collection and logical ID dissemination.** Once the communication channel has been built by CBP, the physical topology collection and logical ID dissemination over the communication channel can be performed by using PCP and LDP. Essentially, the topology collection is a bottom-up process that starts from leaf devices and blooms up to DAC manager, while the logical ID dissemination is a top-down style that initiates from DAC manager and flows down to the leaf devices.

In PCP, each node reports its node device ID and all its neighbors to its parent node. After receiving all information from its children, an intermediate node merges them (including its own neighbor information) and sends them to its parent node. This procedure continues until DAC manager receives the node and link information of the whole network, and then it constructs the physical network topology. In LDP, the procedure is reverse to PCP. DAC manager sends the achieved device-to-logical ID mapping information to all its neighbor nodes, and each intermediate node delivers the information to its children. Since a node knows the descendants from each child via PCP, it can divide the mapping information on a per-child base and deliver the more specific mapping information to each child. Note that the messages exchanged in both PCP and LDP are uni-cast messages which require acknowledgements for reliability.

### 5.2 Testbed Setup and Experiment

We designed and implemented DAC as an application over the Windows network stack. This application implements the modules described in Section 2: including device-to-logical ID mapping, communication channel building, physical topology collection and logical ID dissemination. We built a testbed using 64 Dell servers and 16 8-port DLink DGS-1008D Gigabit Ethernet switches. Each server has an Intel 2GHz dual-core CPU, 2GB DRAM, 160GB disk and an Intel Pro/1000PT dual-port Ethernet NIC. Each link works at Gigabit.

The topology of our testbed is a BCube(8,1), it has two dimensions and 8 servers on each dimension connected by an 8-port Ethernet switch. Each server uses two ports of its dual-port NIC to form a BCube network. Figure 9 illustrates the physical testbed topology and its corresponding blueprint graph. Note that we only programmed our DAC design on servers, and we did not touch switches in this setup because these switches cannot be programmed. Thus, the blueprint graph of our testbed observed at any server should have a degree of 14 instead of 2 as there are 7 neighbors for each dimension. This server-only setup is designed to demonstrate that DAC works in real-world systems, not its scalability.

In this setup, our DAC application is developed to automatically assign the BCube ID for all the 64 servers in the testbed. A server

| *c-int* | $T$ | CCB | timeout | TC | mapping | LD | **Total** |
|---|---|---|---|---|---|---|---|
| 0 | 30 | 2.8 | 90 | 0.7 | 89.6 | 1.3 | **184.4** |
| 10 | 30 | 26.5 | 90 | 0.4 | 93.3 | 1.5 | **211.7** |
| 0 | 50 | 2.9 | 150 | 0.9 | 90.8 | 1.3 | **245.9** |
| 10 | 50 | 26.4 | 150 | 0.5 | 97.6 | 1.2 | **275.7** |

**Table 2: Time (ms) consumed during autoconfiguration**

is selected as DAC manager by setting its level to 0. To inspect the working process of DAC, we divide DAC into 5 steps and check each of them: 1) CCB (communication channel building): from DAC manager broadcasts the message with level 0 to the last node in the network gets its level, 2) timeout: there is no change in neighboring nodes for $3 * T$ at leaf nodes, 3) TC (physical topology collection): from the first leaf node sends out its TCM to DAC manager receives the entire network topology, 4) mapping: device-to-logical ID mapping time including the I/O time, 5) LD (logical IDs dissemination): from DAC manager sends out the mapping information to all the devices get their logical IDs. Table 2 shows the result with different *c-int* and *T* parameters. Note that *c-int* is to control the number of CBM messages and *T* is the timeout value for CBP broadcast, and $3 * T$ is for TCM triggering. The experiments show that the total configuration time is mainly dominated by the mapping time and $3 * T$, and *c-int* can control and reduce the bustiness of CBM messages. In all the cases, our autoconfiguration process can be done within 300ms.

## 6. PERFORMANCE EVALUATION

In this section, we evaluate DAC via extensive simulations. We first introduce the evaluation methodology and then present the results.

### 6.1 Evaluation Methodology

**Structures for evaluation.** We evaluate DAC via experiments on 4 well-known data center structures: BCube [7], FatTree [8], VL2 [9] and DCell [6]. Among these structures, BCube is the most symmetric, followed by FatTree, VL2, and DCell. DCell is the most asymmetric. All the structures can be considered as sparse graphs with different sparsity. VL2 is the sparsest, followed by FatTree, DCell, and BCube. For each of them, we vary the size as shown in Table 3. Please refer to these papers for details. Since BCube is specifically designed for a modular data center (MDC) sealed in shipping containers, the number of devices in BCube should not be very large. We expect them to be in the thousands, or at most tens of thousands. For FatTree and VL2, we intentionally make their sizes to be as large as hundreds of thousands of nodes. DCell is designed for large data centers. One merit of DCell is that the number of servers in a DCell scales doubly exponentially as the level increases. For this reason, we check the performance of DAC on very large DCell graphs. For example, DCell(6,3) has more than 3.8 million nodes!

**Metrics.** There are 3 metrics in our evaluation. First, we measure the speed of $O_2$ on the above structures, which includes both mapping from scratch (*i.e.*, for brand-new data centers) and mapping for incremental expansion (*i.e.*, for data center expansion). This metric is used to show how efficient $O_2$ is as a device-to-logical ID mapping engine. Then, we estimate the total time DAC takes for a complete autoconfiguration process. Lacking a large testbed, we employ simulations. Last, we evaluate the accuracy of DAC in detecting malfunctions via simulations. All the experiments and simulations are performed on a Linux sever with an Intel 2.5GHz dual-core CPU with 8G DRAM. The server runs Red-Hat 4.1.2 with Linux kernel 2.6.18.

### 6.2 Efficiency of $O_2$ Mapping Engine

**Mapping from scratch.** We study the performance of $O_2$ together with the seminal GI tool proposed in [15] called `Nauty` and another algorithm proposed in digital design automation field called `Saucy` [16]. For `Nauty`, we use the latest version v2.4. For `Saucy`,

| BCube$(n,k)$ | FatTree$(n)$ | VL2$(n_r,n_p)$ | DCell$(n,k)$ |
|---|---|---|---|
| B(4,4)=2304 | F(20)= 2500 | V(20,100)= 52650 | D(2,3)=2709 |
| B(5,4)=6250 | F(40)=18000 | V(40,100)= 102650 | D(3,3)=32656 |
| B(6,4)=14256 | F(60)=58500 | V(60,100)= 152650 | D(4,3)=221025 |
| B(7,4)=28812 | F(80)=136000 | V(80,100)= 202650 | D(5,3)=1038996 |
| B(8,4)=53248 | F(100)= 262500 | V(100,100)=252650 | D(6,3)=3807349 |

**Table 3: Number of devices in each structure.**

it does not calculate the one-to-one mapping nor does the isomorphism check between two graphs by default. Instead, it is a tool to calculate the automorphisms in a graph. We observe that, when inputting two graphs as one bigger graph into `Saucy`, among all the output automorphisms there exist at least one that maps each node in one graph to a node in another given that the two graphs are isomorphic to each other. To compare with `Saucy`, we improve its algorithm to check and calculate a one-to-one mapping between two graphs and call it `Saucy+`. Essentially, `Nauty` includes *candidate pruning via orbit*, `Saucy+` is built on top of `Nauty` and introduces *selective splitting*, and $O_2$ is further built on top of `Saucy+` and includes *candidate selection via SPLD*, as we show in Table 4.

Figure 10 plots the results for device-to-logical ID mapping. Note that, we do not include the I/O time for reading graphs into memory. From the figure, we can see that the mapping time of $O_2$ scales in proportion to the total number of devices in the network.

The results in Figure 10 clearly demonstrate that $O_2$ is faster than both `Nauty` and `Saucy+` on all the evaluated structures. $O_2$ can perform the mapping for all the structures within 10 seconds. More specifically, for BCube(8,4), $O_2$ can finish the mapping in less than 1.5 seconds; for FatTree(100) and VL2(100, 100), $O_2$ needs 4.16 and 1.07 seconds respectively; for DCell(6,3) with 3.8+ million nodes, $O_2$ needs only 8.88 seconds. This finding is not surprising since $O_2$ improves over `Saucy+` and `Nauty`. Note that `Nauty` does not show up in the figures of FatTree, VL2, and DCell, because its run-time for any graph bigger than DCell(3,3), FatTree(40) and VL2(20,100) is too long (*i.e.*, days) to fit into the figures nicely.

To better understand why $O_2$ performs best, we assess the relative effectiveness of the three heuristics used in the algorithms on popular data center structures. We make the following three observations.

First, we find that *candidate pruning via orbit* is very efficient for symmetric structures. For example, `Nauty` needs only 0.07 for BCube(4,4) with 2034 devices, whereas it requires 312 seconds for FatTree(20) with 2500 devices. Another example is that while it only takes less than 8 seconds to perform the mapping for BCube(8,4) with 53248 devices, it fails to obtain the result for either FatTree(40) with 58500 devices or VL2(20,100) with 52650 devices within 24 hours. One factor contributing to this effect is that BCube is more symmetric than either FatTree or VL2 structure.

Second, our experiments suggest that *selective splitting* introduced in `Saucy` should be more efficient for sparse graphs. For example, VL2(100,100) and FatTree(100) have similar numbers of devices (250000+), but VL2 needs only 6.33 seconds whereas FatTree needs 18.50 seconds. This is because VL2(100,100) is sparser than FatTree(100). We have checked the average node degree of these two structures. The average degree for VL2(100,100) is approximately 1.03. Compared with VL2(100,100), FatTree(100) has an average node degree of 2.86, more than 2 times denser.

Finally, when *candidate selection via SPLD* is further introduced in $O_2$ to work together with the above two heuristics, it exhibits different performance gains on different structures. SPLD works best for asymmetric graphs. For example, compared with `Saucy+`, $O_2$, which has the SPLD heuristic, improves the time from 2.97 to 1.31 seconds (or 2.27 times) for BCube(8,4), from 18.5 to 4.16 seconds (or 4.34 times) for FatTree(100), from 6.33 to 1.07 seconds (or 5.92 times) for VL2(100,100), whereas it reduces the time from 44603 to

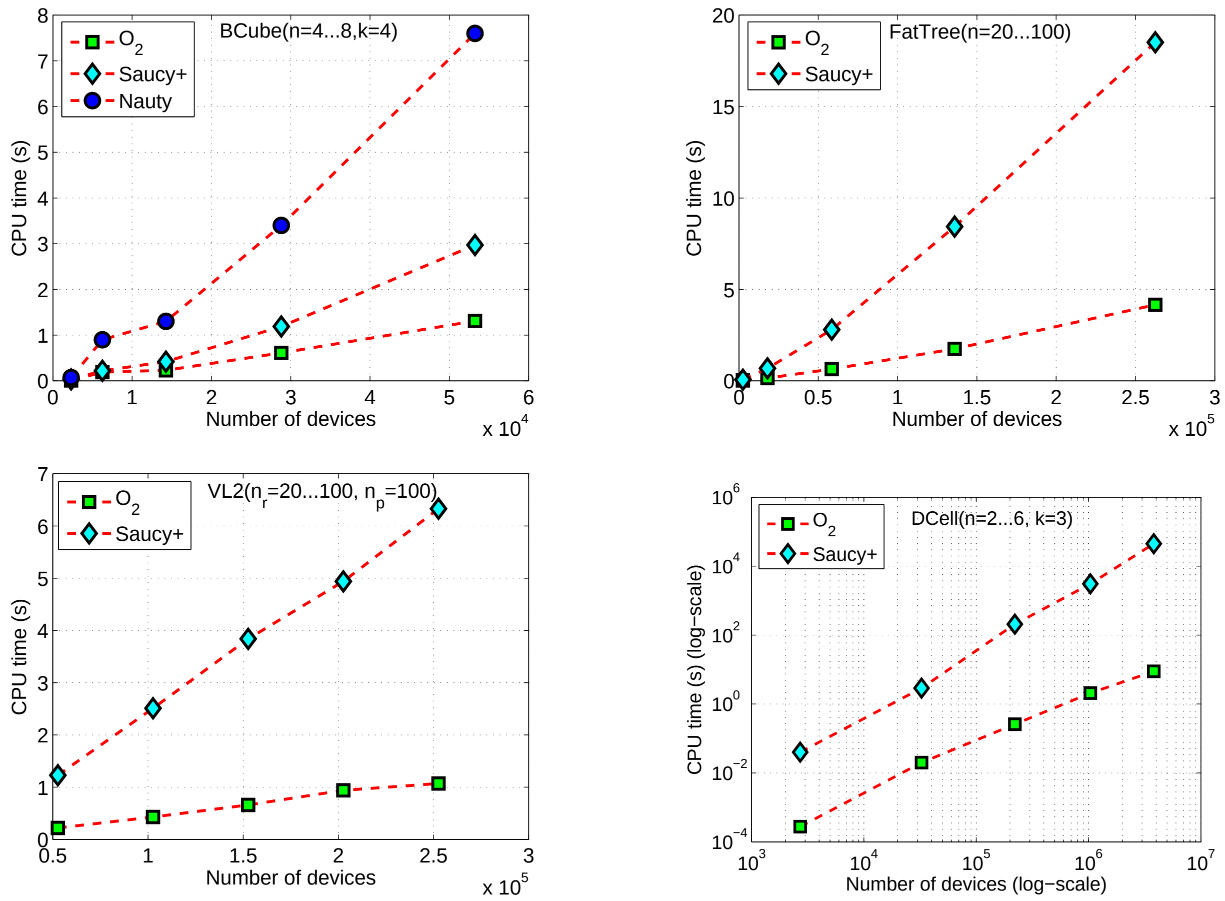

**Figure 10: The speed of $O_2$ one-to-one mapping on BCube, FatTree, VL2 and DCell structures, and its comparison with Nauty and Saucy+. Note that we do not include the performance curves of Nauty on DCell, FatTree and VL2 structures because the run-time of Nauty on all the graphs bigger than DCell(3,3), FatTree(40) and VL2(20,100) respectively is more than one day. Furthermore, we use *log-log* scale to clearly show the performance of both $O_2$ and Saucy+ on DCell.**

| | `Nauty` | `Saucy+` | $O_2$ |
|---|---|---|---|
| Candidate pruning via orbit | √ | √ | √ |
| Selective splitting | | √ | √ |
| Candidate selection via SPLD | | | √ |

**Table 4: Heuristics applied in different algorithms.**

| Old data center | Expanded data center | #Increased devices | Time(s) |
|---|---|---|---|
| BCube(8,3) | BCube(8,4) | 47104 | 0.190 |
| *partial* FatTree(100) | *complete* FatTree(100) | 125000 | 0.270 |
| VL2(50,100) | VL2(100,100) | 125000 | 0.240 |
| DCell(6,2) | DCell(6,3) | 3805242 | 7.514 |

**Table 5: CPU time of mapping for data center expansion.**

8.88 seconds (or 5011 times) for DCell(6,3)! This is because the more asymmetric a graph is, the more likely that the SPLDs of two nodes will be different. In our case, BCube is the most symmetric structure since all the switches are interchangeable, whereas DCell is the most asymmetric one since there are only two automorphisms for a DCell.

We have also checked other combinations of the heuristics, including *selective splitting*, *candidate pruning via orbit* plus *candidate selection via SPLD*, and *selective splitting* plus *candidate selection via SPLD*. We omit the details due to space constraints. The results of all these combinations confirm the above observations: *candidate pruning via orbit* is efficient for symmetric graphs, *selective splitting* works well for sparse graphs, and *candidate selection via SPLD* improves both heuristics and has remarkable performance gain for asymmetric graphs such as DCell.

**Mapping for Incremental Expansion.** For the evaluation of $O_2$ on incremental expansion, we choose one expansion scenario for each structure. Since BCube and DCell are recursively defined, we expand them by increasing the level. For FatTree and VL2, we expand them by increasing the number of servers in each rack. The results are listed in Table 5. We find that all the mappings can be done efficiently. For BCube, we extend BCube(8,3) to BCube(8,4) and finish the mapping in 0.19 second; For FatTree, we expand *partial* FatTree(100), where each edge switch connects to 25 servers, to *complete* FatTree(100), where each edge switch connects to 50 servers, and take 0.47 second for mapping; For VL2, we expand VL2(50,100) to VL2(100,100) and spend 0.24 second; For DCell, we extend DCell(6,2) to DCell(6,3) and use 7.514 seconds. Finally, we check and verify that $O_2$ keeps logical IDs for old devices unmodified.

## 6.3 Estimated Time Cost on Autoconfiguration

Recall that, in Section 5, we have evaluated the time cost of DAC on our BCube(8,1) testbed. In this section, we estimate this time on large data centers via simulations. We use the same parameters *c-int* (checking interval) and $T$ (timeout for CBP broadcast) as in the implementation, and set *c-int* as 10ms and $T$ 50ms. We estimate the time cost for each of the five phases, *i.e.*, CCB, timeout, TC, mapping and LD, as described in Section 5. In the simulations, *device ID* is a 48-bit MAC address and *logical ID* is set to 32 bits, like an IP

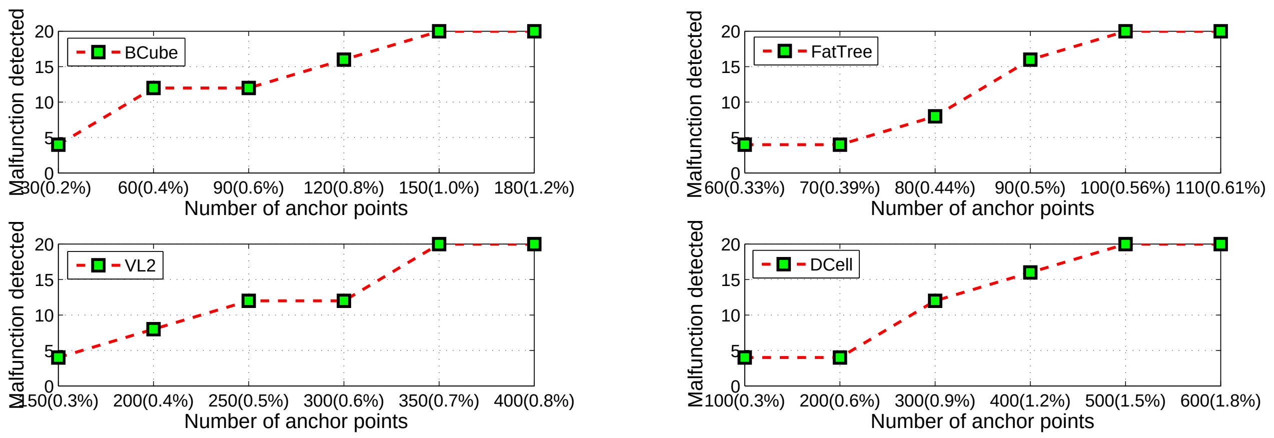

**Figure 11: Number of malfunctioning devices detected with increased number(percent) of selected anchor points.**

| | CCB | timeout | TC | mapping | LD | **Total** |
|---|---|---|---|---|---|---|
| BCube(4,4) | 120 | 150 | 10 | 20 | 10 | **310** |
| BCube(8,4) | 120 | 150 | 10 | 1710 | 10 | **2000** |
| FatTree(20) | 80 | 150 | 6 | 20 | 6 | **262** |
| FatTree(100) | 80 | 150 | 17.6 | 5950 | 26 | **6223.6** |
| VL2(20,100) | 80 | 150 | 7.5 | 360 | 9.2 | **606.7** |
| VL2(100,100) | 80 | 150 | 17.1 | 1760 | 25.2 | **2032.3** |
| DCell(2,3) | 170 | 150 | 15 | 3 | 15 | **353** |
| DCell(6,3) | 250 | 150 | 82.8 | 44970 | 125.3 | **45578.1** |

**Table 6: Estimated time (*ms*) of autoconfiguration.**

address. We assume all the links are 1G/s and all communications use the full link speed. For each structure, we choose the smallest and largest graphs in Table 3 for evaluation. The results are shown in Table 6. From the table, we find that, except DCell(6,3), the autoconfiguration can be finished in less than 10 seconds. We also find that, for big topologies like BCube(8,4), DCell(6,3), FatTree(100) and VL2(100,100), the mapping time dominates the entire autoconfiguration time. DCell(6,3) takes the longest time, nearly 45 seconds, to do the mapping. While the CPU time for the mapping is only 8.88 seconds, the memory I/O time is 36.09 seconds. Here we use more powerful Linux servers than what we used in the implementation, so the mapping here is relatively faster than that in Section 5.

## 6.4 Results for Malfunction Detection

Since malfunctions with degree change can be detected readily, in this section we focus on simulations on the miswirings where there is no degree change. We evaluate the accuracy of our algorithm proposed in Figure 8 in detecting such malfunction. Our simulations are performed on all 4 structures. For each one, we select a moderate size with tens of thousands of devices for evaluation, specifically, they are BCube(6,4), FatTree(40), VL2(20,100) and DCell(3,3). As we know, miswirings without degree change are exceedingly rare and every such case requires at least 4 miswired devices. So in our simulations, we randomly create 5 groups of such miswirings with a total of 20 miswired nodes. In the output of our algorithm, we check how many miswired nodes we have detected versus the number (or percent) of anchor points we have selected. We say a miswired node is detected only if there is no normal node above it in the counter list. This is because the administrators will rectify the miswirings according to our list sequentially and stop once they come to a node that is not really miswired.

Figure 11 demonstrates the results. It clearly shows that the number of detected malfunctions is increased with the number of selected anchor points. In our experiments on all structure, we can detect all the malfunctions with at most 1.5% of nodes selected as anchor points. Interestingly, we find the counter values of good nodes and those of bad nodes are well separated, *i.e.*, there is a clear *drop* in the sorted counter value list. We also find that for different structures, we need different numbers of anchor points in order to detect all 20 miswired devices. For example, in DCell we require as many as 500 pairs of nodes as anchor points to detect all the malfuctions; and in VL2, we need 350 pairs of nodes to detect them all. However, in BCube and FatTree, we only need 150 and 100 anchor points, respectively, to detect all malfunctions. One reason for the difference is that our selected DCell and VL2 networks are larger than BCube and FatTree. Another reason is that different structures can result in different false positives in *Anchor_Pair_Selection()*.

At last, it is worth mentioning that the above malfunction detection has been done efficiently. In the worst case, we used 809.36 seconds to detect all the 20 malfunctioning devices in DCell from 500 anchor points. Furthermore, as mentioned before, the calculations starting from different anchor points are independent of each other, and can be performed in parallel for further acceleration.

## 7. RELATED WORK

In this section, we review the work related to DAC. The differences between DAC and other schemes in related areas such as Ethernet and IP networks are caused by different design goals for different scenarios.

**Data Center Networking.** Portland [8] is perhaps the most related work to DAC. It uses a distributed location discovery protocol (LDP) for PMAC (physical MAC) address assignment. LDP leverages the multi-rooted tree topology property for switches to decide their levels, since only edge switches directly connect to servers. DAC differs from Portland in several aspects: 1) DAC can be applied to arbitrary network topologies whereas LDP only works for multi-rooted tree. 2) DAC follows a centralized design because centralized design significantly simplifies the protocol design in distributed systems, and further, data centers are operated by a single entity.

**Plug-and-play in Ethernet.** Standing as one of the most widely used networking technologies, Ethernet has the beautiful property of "plug-and-play". It is essentially another *notion* of autoconfiguration in that each host in an Ethernet possesses a persistent MAC address and Ethernet bridges automatically learn host addresses during communication. Flat addressing simplifies the handling of topology dynamics and host mobility with no human input to reassign addresses. However, it suffers from scalability problems. Many efforts, such as [25–27], have been made towards a scalable bridge architecture. More recently, SEATTLE [28] proposes to distribute ARP state among switches using a one-hop DHT and makes dramatic advances toward a plug-and-play Ethernet. However, it still cannot well support large data centers since: 1)switch state grows with end-hosts; 2) routing needs all-to-all broadcast; 3) forwarding loop still exists [8].

**Autoconfiguration in IP networks.** Autoconfiguration protocols for traditional IP networks can be divided into stateless and stateful approaches. In stateful protocols, a central server is employed to record state information about IP addresses that have already been assigned. When a new host joins, the severs allocate a new, unused IP to the host to avoid conflict. DHCP [3] is a representative protocol for this category. Autoconfiguration in stateless approaches does not rely on a central sever. A new node proposes an IP address for itself and verifies its uniqueness using a duplicate address detection procedure. For example, a node broadcasts its proposed address to the network, if it does not receive any message showing the address has been occupied, it successfully obtains that address. Examples include IPv6 stateless address autoconfiguration protocol [29] and IETF Zeroconf protocol [30]. However, neither of them can solve the autoconfiguration problem in new data centers where addresses contain locality and topology information.

## 8. CONCLUSION

In this paper, we have designed, evaluated and implemented DAC, a generic and automatic Data center Address Configuration system. To the best of our knowledge, this is the first work in address autoconfiguration for generic data center networks. At the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC has made an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem, and solved it in low time-complexity by leveraging the sparsity and symmetry (or asymmetry) of data center structures. The DAC malfunction detection scheme is able to detect various errors, including the most difficult case where miswirings do not cause any node degree change.

Our simulation results show that DAC can accurately find all the hardest-to-detect malfunctions and can autoconfigure a large data center with 3.8 million devices in 46 seconds. In our implementation on a 64-server BCube testbed, DAC has used less than 300 milliseconds to successfully autoconfigure all the servers. Our implementation experience and experiments show that DAC is a viable solution for data center network autoconfiguration.

## Acknowledgments

We thank Geoffry Nordlund, Wei Shen, and Yaohui Xu for telling us how IP addresses are assigned within MSRA research data center; Guohan Lu for testbed setup; Junfeng Zhou for his help on educating us the operation and miswiring issues in large commercial data centers; Xi Chen for helpful discussion on the complexity of the general graph isomorphism problem. We thank the anonymous reviewers for their constructive comments and our shepherd Michael Mitzenmacher for his insightful and detailed feedback and suggestions, which improve the content and presentation of this paper.

## 9. REFERENCES

[1] R. H. Katz, "Tech Titans Building Boom," *IEEE SPECTRUM*, Feb 2009.

[2] L. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, March 2003.

[3] R. Droms, " Dynamic Host Configuration Protocol," *RFC 2131*, March 1997.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP*, 2003.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[6] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A Scalable and Fault Tolerant Network Structure for Data Centers," in *SIGCOMM*, 2008.

[7] 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," in *SIGCOMM*, 2009.

[8] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric," in *SIGCOMM*, 2009.

[9] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *SIGCOMM*, 2009.

[10] [Online]. Available: http://royal.pingdom.com/2007/10/30/human-errors-most-common-reason-for-data-center-outages/

[11] Z. Kerravala, "As the value of enterprise networks escalates, so does the need for configuration management," *The Yankee Group*, Jan 2004.

[12] Juniper, "What is behind network downtime?" 2008.

[13] [Online]. Available: http://searchdatacenter.techtarget.com/news/column/0,294698,sid80_gci1148903,00.html

[14] "Graph isomorphism problem," http://en.wikipedia.org/wiki/Graph_isomorphism_problem.

[15] B. D. McKay, "Practical graph isomorphism," in *Congressus Numerantium*, 1981.

[16] P. T. Darga, K. A. Sakallah, and I. L. Markov, "Faster Symmetry Discovery using Sparsity of Symmetries," in *45st Design Automation Conference*, 2008.

[17] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu, "FiConn: Using Backup Port for Server Interconnection in Data Centers," in *Infocom*, 2009.

[18] E. M. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time," in *Journal of Computer and System Sciences*, 1982.

[19] "Graph automorphism," http://en.wikipedia.org/wiki/Graph_automorphism.

[20] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting Structure in Symmetry Generation for CNF," in *41st Design Automation Conference*, 2004.

[21] "Data Center Network Overview," *Extreme Networks*, 2009.

[22] "Maximum common subgraph problem," http://en.wikipedia.org/wiki/Maximum_common_subgraph_isomorphism_problem.

[23] V. Kann, "On the approximability of the maximum common subgraph problem," *Annual Symposium on Theoretical Aspects of Computer Science*, 1992.

[24] "Personal communications with opeartor of a large enterprise data center," 2009.

[25] T. Rodeheffer, C. Thekkath, and D. Anderson, "SmartBridge: A scalable bridge architecture," in *SIGCOMM*, 2000.

[26] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: scaling Ethernet to a million nodes," in *HotNets*, 2004.

[27] R. Perlman, "Rbridges: Transparent routing," in *Infocom*, 2004.

[28] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: a scalable ethernet architecture for large enterprises," in *SIGCOMM*, 2008.

[29] S. Thomson and T. Narten, "IPv6 Stateless Address Autoconfiguration," *Expired Internet Draft*, December 1998.

[30] S. Cheshire, B. Aboba, and E. Guttman, "Dynamic configuration of IPv4 link-local addresses," *IETF Draft*, 2003.