Radar
Reliable Resource Scheduling for Composable/Disaggregated Data Centers
Guo, Chao; Zukerman, Moshe; Wang, Tianjiao

Published in:
IEEE Transactions on Industrial Informatics

Published: 01/08/2023

Document Version:
Post-print, also known as Accepted Author Manuscript, Peer-reviewed or Author Final version

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1109/TII.2022.3222348

Publication details:

Citing this paper
Please note that where the full-text provided on CityU Scholars is the Post-print version (also known as Accepted Author Manuscript, Peer-reviewed or Author Final version), it may differ from the Final Published version. When citing, ensure that you check and use the publisher's definitive version for pagination and other details.

General rights
Copyright for the publications made accessible via the CityU Scholars portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights. Users may not further distribute the material or use it for any profit-making activity or commercial gain.

Publisher permission
Permission for previously published items are in accordance with publisher's copyright policies sourced from the SHERPA RoMEO database. Links to full text versions (either Published or Post-print) are only available if corresponding publishers allow open access.

Take down policy
Contact lbscholars@cityu.edu.hk if you believe that this document breaches copyright and provide us with details. We will remove access to the work immediately and investigate your claim.
Abstract—Hardware disaggregation decouples resources (e.g., processors and memory) from monolithic servers, potentially improving service reliability. However, from another perspective, directly exposing resource modules to a shared network may adversely affect service reliability. In this paper, we study a reliable resource allocation problem in disaggregated DCs (DDCs), considering network impact and different disaggregation scales. We provide a mixed-integer linear programming formulation and a resource allocation framework named Radar for this problem. Numerical results demonstrate that the benefits of hardware disaggregation may be adversely affected by an imperfect network. It also shows that both the hardware backup and a proposed migration-based restoration can be applied to overcome this potential adverse effect.

Index Terms—Composable/disaggregated infrastructure, hardware disaggregation, data center, reliability, network

I. INTRODUCTION

With the rapid growth of internet technology like big data and cloud/fog/edge computing, vast amounts of data are poured into the data center (DC), imposing a significant burden on data centers [1], [2]. Considerable effort has been made to ease this burden through various aspects, including resource allocation [3-5], energy efficiency [6-9], and thermal issues [10], [11], which are for server-based architecture. This architecture has caused significant resource stranding, hindering efficiency improvement [12].

Composable/disaggregated infrastructure (CDI), which “uses an API to create physical systems from shared pools of resources” [13], is an emerging computing infrastructure for future data centers (DCs) to break through the boundary limits of traditional servers [14], [15]. In its current early stage, CDI has grown fast due to its high benefits, which is expected to grow to $13.5 billion by 2027 with an annual growth rate of 21 percent [16]. This paper refers to a CDI-based DC as a disaggregated DC (DDC), which represents a DC employing hardware disaggregation. Hardware disaggregation decouples resources (e.g., processors and memory) from integrated servers and reassembles them into resource pools interconnected through a fast network, converting a server-based DC (SDC) into a DDC. New techniques like non-volatile memory express [13], compute express link [13], computational storage [17], and the advancement in optical interconnection [18-20] provide diversified support for communication solutions to hardware disaggregation. DDCs achieve resource efficiency and flexibility by reducing resource stranding in the SDC [12], [21-25]. Hardware upgrades and resource expansion become cost-efficient since they can be operated at the component level instead of the server level [22].

In this paper, we focus on service reliability in DDCs. Providing high service reliability is critical for DCs to provide continuous operations, ensuring high quality of services, while unreliable service may lead to severe economic loss [26]. Compared to SDCs, resource availability and reliability in DDCs are potentially improved for two main reasons. Firstly, improved flexibility expands the optimization regions [27], and secondly, resource decoupling leads to a less harmful failure pattern where failures among different resources may not implicate each other as in SDCs [28]. However, disaggregation may also adversely affect service reliability as resource modules are now directly exposed to a shared network. The failure of the shared network may lead to the unavailability of many resource modules.

In addition, due to the strict latency and bandwidth requirements of inter-resource communications, e.g., CPU-memory communication, the scale of disaggregation is limited [29]. Most efforts on resource disaggregation have considered rack-scale, where a resource, e.g., CPU, can use a different resource, e.g., memory, from the same rack but not from a different rack. Although several publications considered pod/DC-scale disaggregation [29], [30], their practical application is limited to only a few cases.

In this paper, we study the reliability performances in DDCs considering network impact and different disaggregation scales. We summarize the key novelty and main contributions of this paper as follows.

- We study the problem of reliable resource allocation for DDCs considering network effects and different disaggregation scales. In addressing this problem, we aim to achieve both high resource efficiency and high reliability. We consider both static and dynamic scenarios. For the static scenario, the resources are allocated to a batch of known requests. We aim to maximize the reliability of each accepted request and the acceptance ratio, defined as the ratio of the number of accepted requests to the total number of requests. For the dynamic scenario, where requests arrive and depart randomly,
resources are allocated to each request upon its arrival and are released at its departure. In addition, hardware failures occur over time, and each failure is fixed after a certain time to repair. Hardware failures interrupt their hosted requests, resulting in the requests failing to complete their service. The objectives in the dynamic scenario are to minimize the blocking probability (one minus the acceptance ratio) and the number of accepted requests failing to complete their service.

- We provide mixed-integer linear programming (MILP) formulations to solve the multi-objective problem in the static scenario by converting it into a single-objective problem by the weighted sum approach. We first provide a MILP formulation for a DC-scale DDC and then extend it to MILP formulations for an SDC and a rack/pod-scale DDC. We provide approximate Pareto fronts by solving the MILP with varied weights in the objective function.

- We propose a heuristic algorithm for virtual machine (VM) placement for a rack-scale DDC to demonstrate that resource allocation method for request scheduling to maximize the availability of the application and minimize the total cost when allocating resources. However, this work is different from our work. They considered that a user requires a specific number of resource components of each type, e.g., the requirement of three CPU chips and two memory modules. By comparison, we allow one component to be used by multiple requests, and each request comes with a given resource demand, e.g., 10 GB of memory demand. The second work was done by Guo et al. [27], who considered the resource allocation form similar to ours. They provided a resource allocation method for request scheduling to maximize the acceptance ratio subject to meeting requests’ requirements. However, both [27] and [35] assumed a resilient network and a DC-scale disaggregation, which may be unrealistic, and a DC-scale DDC may not be able to support some applications because of latency and bandwidth requirements of inter-resource communications. Different from them, we consider different disaggregation scales and study the impact of a network that is not resilient. We also consider the latency and bandwidth requirement of inter-resource communications, which they did not consider.

III. OVERVIEW OF SDC VS. DDC ARCHITECTURES

In this section, we introduce the DDC architectures we consider for the reliable resource allocation problem (which is defined in the next section). We also discuss the reliability challenges of these DDC architectures.

Fig. 1 (a) shows an SDC architecture, where computing and storage resources are packaged in server blades, each containing different resources, and the onboard circuit provides communications among different resources. These servers are interconnected through a DC network (DCN). Traditional DCNs are interconnected through wired links and are organized in a tree-like topology, e.g., leaf-spine [36] topology, like the one shown in Fig. 1(a). Recently, wireless communications
techniques have also been introduced to reduce the reconfiguration complexity [37].

IV. PROBLEM STATEMENT AND REQUEST RELIABILITY

A. Problem Description

We define the problem of resource allocation in DDCs as follows. We are given a DDC as an undirected graph consisting of nodes and links, and the nodes include switches and blades (See Fig. 1). We assume that DDCs employ optical circuit switches because they provide ultra-low and deterministic latency and guaranteed network bandwidth [18], [39], [40]. Each switch is characterized by a switching delay and reliability. Each blade contains a set of components, e.g., CPU components (See Fig. 1), and each component is characterized by its resource type, available capacity, and reliability. Like Carbonari et al. [28], we assume that different components in a blade fail independently. Each blade is also associated with delay and reliability, which are the delay between receiving and transmitting data and the reliability of the peripheral elements, such as interfaces and control circuits. Each link is characterized by available bandwidth, propagation delay, and reliability.

We use the term request to represent a possible form of resource allocation requests, including requests for jobs/tasks [4], virtual machines [41], [42], and virtual containers [43]. We consider the static and dynamic scenarios of problems regarding whether the requests arrive at once or randomly. Each request arrives with a given service time and a resource demand for each resource type. The request’s inter-resource traffic demand and latency requirements are also given. As in [25], [31], [32], we assume that a request can only use one blade for one resource, and the disaggregation scale determines whether it can obtain different resources from different blades. Consider the rack-scale DDC as an example, where a request can only use memory from one blade but obtains different resources from multiple blades within one rack. Existing work such as [25], [31], [32] regards a blade in a DDC as a single node but does not consider components in the blade. We also consider how the resources from these components are allocated because we need to consider the failure independence among different components. Nevertheless, a request can obtain one resource from multiple components in a blade in a DDC.

The problem objectives and methodologies are different in the static and dynamic scenarios.

1) Static Scenario

In this scenario, all requests arrive at once. Our aim in this scenario is to maximize the acceptance ratio and each request’s reliability. We provide a MILP formulation and a scalable heuristic algorithm to address the problem. The MILP formulation translates the problem into precise mathematical language, and it can be solved using commercial solvers to provide optimal solutions. The optimal solutions can be further used to validate the efficiency of the proposed algorithm.

2) Dynamic Scenario

In this scenario, requests arrive and leave randomly and sequentially. A request is accepted if the available resources are sufficient upon arrival and blocked otherwise. We assume no waiting room, and a blocked request is rejected and leaves the system without re-attempting. One of our objectives is to minimize the blocking probability (or maximize the acceptance ratio), defined as the ratio of the number of blocked requests to the total number of arrivals during a specified period. An
accepted request may be interrupted by a hardware failure, failing to complete its service. Our other objective is to minimize the number of accepted requests failing to complete service. The blocking probability does not include the accepted requests interrupted by hardware failures. We provide a Radar framework to achieve the two objectives.

B. The Reliability of a Service Request

Assume that a request (denoted \(i\)) arrives, and denote the arrival and departure times of request \(i\) as \(t_i^a\) and \(t_i^d\), respectively. Also, assume that an element (denoted \(e\)) was last repaired at the time \(t_e^d\). Let the random variable \(\Delta t\) be the time between failures (TBF) of element \(e\). As commonly used, TBF is the time from the moment the element is repaired until it fails again. The probability that \(e\) does not fail during the service time of request \(i\) (denoted \(R_e^i\)) can be obtained by:

\[
P_e^i = P(\Delta t > t_i^d - t_e^d | \Delta t > t_i^a - t_e^d)
\]

where \(R_e^i\) is the reliability of element \(e\), i.e., the probability that the TBF of \(e\) is not shorter than \(t_i^a\).

We consider the reliability of request \(i\), denoted by \(R_i\), as the probability that request \(i\) encounters no hardware failure during its service time. This is equal to the probability that no element that serves it fails during its service time. Assuming independence of failures among different elements, we obtain:

\[
R_i = \prod_{e \in E_i} R_e^i,
\]

where \(E_i\) denotes the set of elements used by request \(i\).

V. THE MILP FORMULATION

A. MILP Formulation for a DC-Scale DDC

We first introduce the MILP for a DDC of DC-scale and later extend it to other scales. Table I provides our used notations.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>Set of resource types, e.g., CPU and memory</td>
</tr>
<tr>
<td>(N)</td>
<td>Set of blades</td>
</tr>
<tr>
<td>(X)</td>
<td>Set of switches</td>
</tr>
<tr>
<td>(NE_m)</td>
<td>Set of nodes neighboring to node (m \in N \cup X)</td>
</tr>
<tr>
<td>(C_n)</td>
<td>Set of (resource) components in blade (n)</td>
</tr>
<tr>
<td>(I)</td>
<td>Set of requests</td>
</tr>
<tr>
<td>(\delta_{nc}^i)</td>
<td>Binary parameter indicating whether component (c) in blade (n) is of resource type (r \in R)</td>
</tr>
<tr>
<td>(A_{nc})</td>
<td>Available capacity of component (c) in blade (n)</td>
</tr>
<tr>
<td>(B_{mn})</td>
<td>Available bandwidth of the link ((m, n))</td>
</tr>
<tr>
<td>(t_{mn}^d)</td>
<td>Delay of blade (n)</td>
</tr>
<tr>
<td>(t_{mn}^s)</td>
<td>Switching delay of switch (n)</td>
</tr>
<tr>
<td>(\delta_{mn}^\gamma)</td>
<td>Propagation delay of the link ((m, n))</td>
</tr>
<tr>
<td>(\gamma_{mn}^\delta)</td>
<td>Binary parameter denoting whether the traffic of ((r_1, r_2)), (r_1, r_2 \in R) is allowed to travel the link (mn)</td>
</tr>
<tr>
<td>(P_{nc})</td>
<td>The probability that component (c) in blade (n) does not fail during the service time of request (i)</td>
</tr>
<tr>
<td>(P_{mn})</td>
<td>The probability that the blade or switch (n) does not fail during the service time of request (i)</td>
</tr>
<tr>
<td>(D_{ir})</td>
<td>Resource demand of request (i) for resource (r)</td>
</tr>
<tr>
<td>(\lambda_{ir}^\delta)</td>
<td>Traffic demand of resource pair ((r_1, r_2)) in request (i)</td>
</tr>
<tr>
<td>(\Delta_{ir}^\gamma)</td>
<td>Latency requirement of traffic ((r_1, r_2)) in request (i)</td>
</tr>
</tbody>
</table>

Note that the requirement is for the extra latency traversing the network relative to the onboard latency in a server.

\(\gamma\)  | A large value |
| \(\alpha\) | A weight factor |

### Decision Variables

\(\pi_i\)  | (Binary) Equal one if request \(i\) is accepted; zero otherwise |
\(\delta_{nc}^i\)  | (Binary) Equal one if request \(i\) gets resource \(r\) from the blade \(n\); zero otherwise |
\(\lambda_{ir}^\delta\)  | (Binary) Equal one if request \(i\) gets both resources \(r_1\) and \(r_2\) from blade \(n\); zero otherwise |
\(\mu_{mn}^\gamma\)  | (Real) The amount of resource \(r\) that request \(i\) gets from component \(c\) in blade \(n\) |
\(\gamma_{mn}^\delta\)  | (Binary) Equal one if the traffic of \((r_1, r_2)\) in request \(i\) traverses link \((m, n)\); zero otherwise |
\(\rho_{in}^\alpha\)  | (Binary) Equal one if the traffic of \((r_1, r_2)\) in request \(i\) traverses switch \(n\); zero otherwise |
\(\rho_{in}^\beta\)  | (Binary) Equal one if request \(i\) uses switch or blade node \(n\) \((n \in N \cup X)\); zero otherwise |
\(\lambda_i\)  | (Real) The logarithm of the reliability of request \(i\), i.e., \(\lambda_i = \log(R_i)\), where \(R_i\) is the reliability of the request |
\(\Lambda_{min}\)  | (Real) The minimum value of all \(\lambda_i\) |

Our objective is to maximize the weighted sum of the minimum request reliability and acceptance ratio, formulated as:

\[
\text{Maximize: } (1 - \alpha) \sum_{i \in I} \frac{\pi_i}{|I|} + \alpha \cdot \Lambda_{min}.
\]  (3)

The decision variables in (3) are defined in Table I. Note that some decision variables in the table are not included in the objective function but are included in the constraints.

- **Blade and component allocation constraints**

\[
\pi_i = \sum_{n \in N} \delta_{nc}^i \forall i \in I, r \in R
\]  (4)

\[
\nabla \cdot \delta_{nc}^i \geq \mu_{nc}^\gamma \forall i \in I, r \in R, n \in N, c \in C_n
\]  (5)

\[
\delta_{nc}^i \leq \sum_{c \in C_n} \mu_{nc}^\gamma \forall i \in I, r \in R, n \in N
\]  (6)

\[
\sum_{n \in N} \mu_{nc}^\gamma \leq A_{nc} \cdot \theta_{nc}^t \forall n \in N, c \in C_n, r \in R
\]  (7)

- **Traffic scheduling constraints**

\[
\sum_{i \in I, r_1, r_2 \in R, r_1 \neq r_2} \gamma_{mn}^{r_1 r_2} \cdot \lambda_{ir}^\delta \leq B_{mn}
\]  (9)

\[
\sum_{m \in N \cup X, n \in N} \gamma_{mn}^{r_1 r_2} \cdot \lambda_{ir}^\delta \leq \Lambda_{mn}^{r_1 r_2} = \sum_{m \in N} \lambda_{m}^{r_1 r_2} - \sum_{m \in N} \lambda_{m}^{r_1 r_2} - \sum_{m \in N} \gamma_{mn}^{r_1 r_2}
\]  (10)

\[
\sum_{i \in I, r_1, r_2 \in R, r_1 \neq r_2} \gamma_{mn}^{r_1 r_2} \leq \gamma_{mn}^{r_1 r_2} \forall m \in X \cup N, n \in N \cup M, i \in I, r_1, r_2 \in R
\]  (11)

\[
\sum_{i \in I, r_1, r_2 \in R} \gamma_{mn}^{r_1 r_2} \leq \sum_{i \in I, r_1, r_2 \in R} \gamma_{mn}^{r_1 r_2} \forall m \in N \cup X, n \in N \cup M
\]  (12)

\[
\gamma_{mn}^{r_1 r_2} + \gamma_{mn}^{r_1 r_2} \leq 1 \forall i \in I, r_1, r_2 \in R, n \in N \cup X, m \in N
\]  (13)

\[
\theta_{mn}^{r_1 r_2} \geq \gamma_{mn}^{r_1 r_2} \forall m \in X, n \in N \cup M, i \in I, r_1, r_2 \in R
\]  (14)

\[
\theta_{mn}^{r_1 r_2} \geq \gamma_{mn}^{r_1 r_2} \forall m \in X, n \in N \cup M, i \in I, r_1, r_2 \in R
\]  (15)
Each resource pair for each request. Constraint (11) ensures that flow-conservation constraint for routing the traffic between wastage of communication resources (e.g., creating an Constraint (13) ensures that each traffic stream can only use one component capacity restriction.

Reliability-related constraints: Constraint (19) ensures that the reliability of each accepted request is correctly calculated. The last term on the right-hand side of (19) is to avoid the reliability of a rejected request being 1, which may distort output information. Constraint (20) ensures that the minimum reliability (log form) is no larger than that of each accepted request. Constraints (21) – (22) ensure that a link is used by a request when it carries the traffic of the request. Constraints (23) – (24) ensure that a switch node is used by a request when it switches the requested traffic. Constraints (25) – (26) ensure that a blade is used by a request when it provisions resources to this request. Constraints (27) – (28) ensure that a component is used by a request when it provisions resources to this request. The complexity of both the number of dominant variables ($y_{mn}^{ir}$) and the dominant number of constraints (11) is given by $O(|\mathcal{I}| \cdot |\mathcal{R}|^2 \cdot |\mathcal{N} \cup \mathcal{X}|^2)$.

B. MILP Formulation for a Blade-Scale DDC (SDC)

The following constraints are added to extend the DC-scale DDC to a blade-scale DDC:

$$\sum_{n \in \mathcal{N}} \rho_{n}^{i} = \pi_{i} \forall i \in \mathcal{I}$$  \hspace{1cm} (29)

$$\gamma_{mn}^{ir} = 0 \forall i \in \mathcal{I}, r_{1}, r_{2}, r_{3} \in \mathcal{R}, m \in \mathcal{N} \cup \mathcal{X}, n \in \mathcal{N}_{m}$$  \hspace{1cm} (30)

$$\chi_{mn}^{i} = 0 \forall i \in \mathcal{I}, r_{1}, r_{2} \in \mathcal{R}, m \in \mathcal{N} \cup \mathcal{X}, n \in \mathcal{N}_{m}$$  \hspace{1cm} (31)

$$\omega_{n}^{i} = 0 \forall i \in \mathcal{I}, r_{1}, r_{2} \in \mathcal{R}, m \in \mathcal{N} \cup \mathcal{X}, n \in \mathcal{N}_{m}$$  \hspace{1cm} (32)

Constraint (29) ensures that each accepted request can only use resources from a single blade, and constraint (30) ensures that each traffic stream does not use network links as it is done locally. Constraints (32) – (33) ensure that an entire request does not use links or switches in an SDC.

C. MILP Formulation for a Rack- or Pod-Scale DDC

Here, a request can only use resources from a single rack (or pod, same as below) in a rack-scale DDC but cannot use resources from different racks. Let $\Gamma$ be the set of racks and $c_{nk}$ be a binary parameter indicating whether the blade $n$ is in rack $k$. Define a binary variable $\beta_{nk}^{k}$ that equals one if request $i$ uses blade in rack $k$; otherwise, zero. The following constraints should be added to extend the MILP for DC-scale DDC to the rack-scale DDC.

$$\sum_{n \in \mathcal{N}} \beta_{nk}^{k} = \pi_{i} \forall i \in \mathcal{I}$$  \hspace{1cm} (34)

$$\beta_{nk}^{k} \geq \rho_{i}^{l} \cdot c_{nk} \forall i \in \mathcal{I}, n \in \mathcal{N}, k \in \Gamma$$  \hspace{1cm} (35)

$$\beta_{nk}^{k} \leq \sum_{n \in \mathcal{N}} \rho_{n}^{i} \cdot c_{nk} \forall i \in \mathcal{I}, k \in \Gamma$$  \hspace{1cm} (36)
that a rack is used by request \( i \) as long as request \( i \) uses blades in the rack.

VI. FRAMEWORK OVERVIEW AND HEURISTIC ALGORITHMS

A. Framework Description

As shown in Fig. 3, the Radar framework consists of a scheduler, monitor, and resource physical modules. The monitor module detects topology and load changes as well as hardware failures and repairs and periodically reports the information to the scheduler module to assist in decision-making. The scheduler module executes appropriate algorithms based on the requests and hardware information to make a decision and finally sends the decision information to the physical DDC for further operation.

B. Resource Allocation Policy

We design indices to quantitatively assess the performance of our resource allocation policies, representing preferences a single request. If the first (and only) batch comprises multiple resources, then it is a dynamic scenario if the first arrival batch comprises one. If the first (and only) batch comprises multiple arrivals, the scheduler assumes it is the static scenario.

\[
\eta_{n}^{RS} = \epsilon \cdot \eta_{n}^{rel} + (1 - \epsilon) \cdot \eta_{n}^{eff} \quad \forall n \in \mathbb{N},
\]

where \( \eta_{n}^{rel} \) and \( \eta_{n}^{eff} \) are the reliability and efficiency indices associated with blade \( n \), respectively, and \( \epsilon \in [0, 1] \) is the weighting coefficient. The reliability index \( \eta_{n}^{rel} \) is the probability that blade \( n \) does not fail during the service time of the request, and the efficiency index \( \eta_{n}^{eff} = \bar{U}_{n} = \sum_{r \in R} U_{n}^{r} / |R| \), where \( U_{n}^{r} \) is the utilization of resource \( r \) in blade \( n \). The efficiency index is set according to the well-known best-fit (BF) bin-packing scheme, which selects a feasible bin with the least remaining capacity [44].

2) Rack- or Pod-Scale DDC

Since the rack- and pod-scale DDCs are similar, we next consider only the rack-scale DDC. The allocation in a rack-scale DDC is to select a rack and then choose a blade for each type of resource, which involves rack and blade indices.

Similar to (37), the blade index is also the weighted sum of efficiency and reliability indices. The efficiency index is the utilization of the blade. Since a DCC blade has multiple components of the same resource type, the utilization of the blade is calculated by \( \sum_{c \in \text{comp}} L_{nc} / \sum_{c \in \text{comp}} A_{nc} \), where \( L_{nc} \) and \( A_{nc} \) are the load and available capacity of component \( c \) in blade \( n \), respectively. The reliability index of a blade is \( p_{1n}^{R} \cdot \prod_{c \in \text{comp}} p_{1c}^{R} \), where \( p_{1n}^{R} \) is the set of components in blade \( n \) used by request \( i \). Note that the reliability index here only considers the components used by the request because components are independent and do not interfere with each other. Overall, the blade index \( \eta_{n}^{RS} \) is:

\[
\eta_{n}^{RS} = \epsilon \cdot p_{1n}^{R} \cdot \prod_{c \in \text{comp}} p_{1c}^{R} \cdot (1 - \epsilon) \cdot \frac{\sum_{c \in \text{comp}} L_{nc} / \sum_{c \in \text{comp}} A_{nc}}{\prod_{c \in \text{comp}} p_{1c}^{R}}.
\]

The rack index is also the weighted sum of a reliability index and an efficiency index. The reliability index of a rack is the request’s reliability when allocated with the selected blades in the rack and the required switches and links. The efficiency index is defined as the average utilization of the \( |R| \) selected blades in the rack. Finally, the rack (or pod) with the highest rack index is chosen.

C. The Algorithm for One Request in a Rack-Scale DDC

1) Algorithm’s Overview

Fig. 4 provides the algorithm pseudocode, named ARRIVAL-ALLOC, for allocating resources to a request. It takes two inputs, i.e., the request \( i \) and the rack-scale DDC graph \( G \). The algorithm first sorts the resource types in \( R \) in descending order of competitive ratio [18], defined as the ratio of the requested resource amount to the capacity per component for each resource type \( r \in R \). This operation prioritizes intensive resources, e.g., for \( r = \text{CPU} \), a CPU-intensive request has the largest competitive ratio. If components have different capacities, the denominator of the competitive ratio is the average component capacity.

<table>
<thead>
<tr>
<th>ARRIVAL-ALLOC (L, G)</th>
<th>// - request, ( G ) - rack-scale DDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: \text{SORT-DES} ( (R, \frac{L_r}{\text{capacity}_r}) )</td>
<td>// sort ( R ) in descending order of ( \frac{L_r}{\text{capacity}_r} )</td>
</tr>
<tr>
<td>2: \text{maxRackIndex} = -1;</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>3: ( \Phi_{best} = \text{Null} );</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>4: for ( (k \in \mathbb{N}) )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>5: for ( (r \in R) )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>6: ( L_{n}^{r} = \emptyset; )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>7: for each ( n ) in ( R )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>8: if (FRAGMENTABLE-BIN-PACK((n, D_{n}))</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>9: ( L_{n}^{r} ), add(n);</td>
<td>// add the feasible blade</td>
</tr>
<tr>
<td>10: if ( \Phi_{best} = \emptyset )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>11: \text{go to line 4; }</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>12: \text{SORT-DES} ( (U_{n}^{r}, \eta_{n}^{rel}) )</td>
<td>// sort the list in increasing order of ( \eta_{n}^{rel} )</td>
</tr>
<tr>
<td>13: ( U_{n}^{r} = \text{sort these lists in one set} )</td>
<td>// sort the list in increasing order of ( \eta_{n}^{rel} )</td>
</tr>
<tr>
<td>14: ( p = 1; )</td>
<td>// set the first resource (type) in ( R )</td>
</tr>
<tr>
<td>15: ( \Phi_{best} = \Phi_{k}; )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>16: if (( \text{BLADE-SELECT-TRAFFIC-sched}(\Phi_{best}, \text{L}, \text{U}, \Phi_{k}) = \text{TRUE} )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>17: if (( \text{RACK-INDEX}(\Phi_{best}) &gt; \text{max RACK-INDEX}(\Phi_{k}) )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>18: \text{max} = \text{RACK-INDEX}(\Phi_{best});</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>19: ( \Phi_{best} = \Phi_{k}; )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>20: if ( \Phi_{best} = \text{Null} )</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>21: \text{FINAL-ALLOC}(\Phi_{best});</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>22: return \text{TRUE; }</td>
<td>// initialize the best solution as empty</td>
</tr>
<tr>
<td>23: return \text{FALSE}</td>
<td>// initialize the best solution as empty</td>
</tr>
</tbody>
</table>

Fig. 4. Pseudocode of the algorithm for allocating resources to a request.

Subsequently, the algorithm scans the racks to find the best solution, i.e., the rack with the maximum rack index. The variable \( \Phi_{best} \) in line 3 is a global variable recording the best solution. This variable has a self-defined data structure that records the information on which components, blades, links, and switches are used for providing the resource to request \( i \). The algorithm updates the current optimal solution when a new
solution with a higher rack index is found (lines 16 – 19). The algorithm iterates until all racks are checked, and finally, the one with the highest rack index is selected (line 21).

In a candidate rack and for each resource type, the algorithm filters the blades with sufficient remaining capacity by calling the procedure FRAGMENTABLE-BIN-PACK (line 8). This procedure addresses the problem of allocating resources from the components in the given blade and outputs TRUE when the blade is feasible to host the request. Afterward, the feasible blades are sorted in descending order of the blade index (line 12) to prioritize those with high blade indices. Next, the algorithm calls the procedure BLADE-SELECT-TRAFFIC-SCHED to find the solution in this rack (line 16), i.e., select blades and schedule the traffic.

2) Allocating Components Resources in a Given Blade

This corresponds to the procedure FRAGMENTABLE-BIN-PACK in Fig. 4. This procedure is based on the algorithm proposed by LeCun et al. [45] to solve the bin-packing problem where the items are splittable. Here, the item to be packed is the resource demand \( D_r \), and the bins are the components in blade \( n \). This procedure operates as follows. Firstly, sort the components in the given blade in decreasing order of \( R_{\text{type}}(n) \) (a probability term defined in Table 1), prioritizing high reliable components. Secondly, try to find out a perfect component [45], i.e., the component whose remaining capacity is precisely equal to \( D_r \). If the perfect component exists (e.g., Request 3 in Fig. 5), assign it to the request and terminate the procedure. Otherwise, check the components one by one. When the remaining capacity of a candidate component is larger than the demand (e.g., Request 1 in Fig. 5), allocate resources from this component. Otherwise, allocate all its remaining capacity to the request and use subsequent components to fulfill the remaining demand (e.g., Request 2 in Fig. 5).

3) Blade Selection and Traffic Scheduling in a Given Rack

BLADE-SELECT-TRAFFIC-SCHED \((p, R, L_k, \phi_k)\)

1: \( r = R[p] \)
2: \( L_k^r = L_k^r \cdot \text{get}(r) \) // the list of blades of resource \( r \)
3: for \( n_{\text{cur}} \in L_k^r \)
4: if (TRAFFIC-SCHED \((\phi_k, n_{\text{cur}})\) == FALSE)
5: continue;
6: \( \phi_k.\text{add}(n_{\text{cur}}) \); // record the feasible blade
7: if \( (p == \{R\}) \)
8: return TRUE;
9: else
10: \( p + 1 \); // index the resource (type) next to \( r \)
11: res = BLADE-SELECT-TRAFFIC-SCHED \((p, R, L_k^r, \phi_k)\)
12: if (res == FALSE)
13: \( \phi_k.\text{remove}(n_{\text{cur}}) \); // failure in resources after \( r \)
14: else
15: return TRUE;
16: return FALSE

This part corresponds to the procedure BLADE-SELECT-TRAFFIC-SCHED in Fig. 4, and Fig. 6 gives the pseudocode of the procedure. The general idea of this procedure is as follows. In the beginning, we select a blade for the first resource type \( (r_{1st}) \). When selecting a blade for the second resource type \( (r_{2nd}) \), we also try to schedule the traffic \( (r_{1st}, r_{2nd}) \). If the traffic scheduling fails, meaning that this blade is not suitable for \( r_{2nd} \), we try the next blade. Similarly, when selecting a blade for the third resource type \( (r_{3rd}) \), we also need to schedule the traffic \( (r_{1st}, r_{3rd}) \) and \( (r_{2nd}, r_{3rd}) \). This process is repeated for each of the remaining resource types.

The procedure takes the input \( p = 1, ..., |R| \), to index the current resource type \( r \), i.e., \( R[p] \) is the \( p^{th} \) element in \( R \). Note that \( R \) is an ordered set that has been sorted previously (see line 12 in Fig. 4). The procedure is executed recursively, starting from the first resource type, i.e., \( p = 1 \). Each a resource type temporarily determines its host, the procedure goes to the next resource type by incrementing \( p \) (line 10). The termination condition is that the procedure finds a feasible blade for the last resource type (lines 7-8).

For a current resource type \( r \) indexed by \( p \), the procedure scans blades in \( L_k^r \) to look for the first blade that satisfies the traffic requirement (line 4). Here, another procedure named TRAFFIC-SCHED is called to schedule the traffic between the current resource type \( r \) and the previous resource types \( r_{\text{prev}} = R[p_{\text{prev}}], p_{\text{prev}} = 1, ..., p - 1 \). If the traffic scheduling succeeds, the procedure temporarily records the current blade \( n_{\text{cur}} \) as the host for \( r \) (line 6), and go for the next resource type \( r_{\text{next}} = R[p + 1] \) (line 10). Then, the procedure recursively calls the algorithm itself (line 11) to find a feasible blade for \( r_{\text{next}} \). If the next resource type cannot find a feasible blade satisfying the traffic requirement, it abandons blade \( n_{\text{cur}} \) (line 13), then tries the next blade for resource type \( r \).

The traffic scheduling procedure TRAFFIC-SCHED searches for a path for each resource pair, which is operated as follows. Firstly, it excludes links with insufficient capacity. Then, it runs the shortest path algorithm with the weight of each link setting as \( w_{mn} = \tau_{mn} + 0.5 \cdot \tau_m + 0.5 \cdot \tau_n \), where, \( \tau_{mn} \), \( \tau_m \), and \( \tau_n \) are the propagation delay of link \( (m, n) \) and the delays at the endpoints \( m \) and \( n \), respectively. If the path exists, it further checks whether the latency meets the requirement and schedules the traffic along this path if it does.

Fig. 7 illustrates the procedure by an example, which includes three resource types, sorted in the order of CPU, memory, and GPU. It starts from the first resource type, i.e., CPU, and checks whether the first CPU blade (C1) is feasible. As it is the first resource type, there are no previous resource types, so no traffic needs to be scheduled, and C1 is directly
recorded as the candidate blade (recorded in $\phi_k$). Then, it goes for the second resource type, i.e., memory, and needs to schedule the traffic between memory and CPU. However, selecting M1 as the candidate leads to failure in traffic scheduling, so M1 is not feasible, and M2 is recorded as a temporary host for the memory. When it comes to the last scheduling, so M1 is not feasible, and M2 is recorded as a

selecting M1 as the candidate leads to failure in traffic when scheduling the traffic between memory and GPU. It returns to the memory blade selection step and abandons M2 but tries M3 and M4. Finally, C1, M4, and G2 are selected, and traffic scheduling is also complete.

4) Algorithm's Complexity

We first analyze the algorithm's time complexity of the algorithm provided in Fig. 4. We consider it takes $O(n \log n)$ to sort a list with $n$ elements, as many sorting algorithms have this complexity. In addition, we assume the number of resource types and the number of components in each blade are constant. Accordingly, lines 4, 8, and 12 take the time of $O(1)$, $O(|I| \cdot |N_{kr}|)$, and $O(|I'| \cdot |N_{kr}| \cdot \log(N_{kr}))$, respectively. Here $|N_{kr}|$ is the number of blades of each resource type per rack. The most time-consuming execution is the blade selection procedure, which is to select one blade for each resource type from a rack. Since the total number of blade combinations is $|N_{kr}|^{n}$, the complexity of the blade selection procedure is $O(|N_{kr}|^{n})$. Therefore, lines 16, 17, and 18 take the time of $O(|I'| \cdot |N_{kr}|^{n})$. In total, the time complexity of the algorithm is $O(|I| \cdot |N_{kr}| \cdot \log|N_{kr}| + |I'| \cdot |N_{kr}|^{n})$. It is equal to $O(|I| \cdot |N_{kr}|^{n})$ since $|R| > 1$. Since $|R|$ has a typical value of 3 to 4, the complexity is very high. Fortunately, the DDC architectures are basically rack-scale, where the base term $|N_{kr}|$ does not have a high value, at most a dozen.

The space complexity of this algorithm is $O(|R|)$. The extra space is needed mainly to recursively call the B LADE-SELECTION procedure.

D. Batch Allocation for the Static Scenario

For this scenario, we introduce the SHUFFLE-AND-UPDATE procedure as follows. Given a set of requests $I$, we first randomize the order of requests in $I$, and then apply the algorithm shown in Fig. 4 to each request according to the order and calculate the objective function, recorded as $F_m$. Nextly, we set an integer number which we refer to as an epoch size. We then repeat the randomize-and-try to find a better solution, i.e., a solution with the objective $F_s$ and $F_s > F_m$. If a better solution can be found after repeating the random-and-try $K$ times, such that $K < \text{epoch size}$, we update the optimal solution with $F_s$. We call this process one epoch of the SHUFFLE-AND-UPDATE optimization process. The above optimization process is further repeated until no better solution is found in any epoch.

E. Migration for the Dynamic Scenario

In order to reduce the number of accepted requests failing to complete their service, we propose a migration mechanism to restore the interrupted requests. Fig. 8 gives the pseudocode of this procedure. The procedure first excludes the failed element from the DDC graph, then, for each interrupted request, calls the procedure ARRIVAL-ALLOC (See Fig. 4) based on the residual graph and re-allocates resources to this request if it succeeds. It should be noted that the failed element can be repaired and can serve other requests after being repaired.

4) Algorithm's Complexity

We first analyze the algorithm's time complexity of the algorithm provided in Fig. 4. We consider it takes $O(n \log n)$ to sort a list with $n$ elements, as many sorting algorithms have this complexity. In addition, we assume the number of resource types and the number of components in each blade are constant. Accordingly, lines 4, 8, and 12 take the time of $O(1)$, $O(|I| \cdot |N_{kr}|)$, and $O(|I'| \cdot |N_{kr}| \cdot \log(N_{kr}))$, respectively. Here $|N_{kr}|$ is the number of blades of each resource type per rack. The most time-consuming execution is the blade selection procedure, which is to select one blade for each resource type from a rack. Since the total number of blade combinations is $|N_{kr}|^{n}$, the complexity of the blade selection procedure is $O(|N_{kr}|^{n})$. Therefore, lines 16, 17, and 18 take the time of $O(|I'| \cdot |N_{kr}|^{n})$. In total, the time complexity of the algorithm is $O(|I| \cdot |N_{kr}| \cdot \log|N_{kr}| + |I'| \cdot |N_{kr}|^{n})$. It is equal to $O(|I'| \cdot |N_{kr}|^{n})$ since $|R| > 1$. Since $|R|$ has a typical value of 3 to 4, the complexity is very high. Fortunately, the DDC architectures are basically rack-scale, where the base term $|N_{kr}|$ does not have a high value, at most a dozen.

The space complexity of this algorithm is $O(|R|)$. The extra space is needed mainly to recursively call the B LADE-SELECTION procedure.

D. Batch Allocation for the Static Scenario

For this scenario, we introduce the SHUFFLE-AND-UPDATE procedure as follows. Given a set of requests $I$, we first randomize the order of requests in $I$, and then apply the algorithm shown in Fig. 4 to each request according to the order and calculate the objective function, recorded as $F_m$. Nextly, we set an integer number which we refer to as an epoch size. We then repeat the randomize-and-try to find a better solution, i.e., a solution with the objective $F_s$ and $F_s > F_m$. If a better solution can be found after repeating the random-and-try $K$ times, such that $K < \text{epoch size}$, we update the optimal solution with $F_s$. We call this process one epoch of the SHUFFLE-AND-UPDATE optimization process. The above optimization process is further repeated until no better solution is found in any epoch.

E. Migration for the Dynamic Scenario

In order to reduce the number of accepted requests failing to complete their service, we propose a migration mechanism to restore the interrupted requests. Fig. 8 gives the pseudocode of this procedure. The procedure first excludes the failed element from the DDC graph, then, for each interrupted request, calls the procedure ARRIVAL-ALLOC (See Fig. 4) based on the residual graph and re-allocates resources to this request if it succeeds. It should be noted that the failed element can be repaired and can serve other requests after being repaired.

F. Discussion on Implementation

The implementation of our proposed resource management framework needs the support of operating systems (OSes) for DDCs, e.g., LegoOS [12] and GiantVM [46]. LegoOS is based on a new OS model, named splithkernel, where each resource component maintains an independent manager (or monitor). LegoOS provides the capability that allows an application to use multiple components of the same resource type, which can be combined with our proposed components allocation procedure. In addition, LegoOS provides fault-tolerance mechanisms, which can be combined with the migration-based restoration algorithm to resume interrupted services. GiantVM is a hypervisor system that can create a VM spanning multiple components and machines (i.e., blades), which may also be used to implement our proposed framework.

VII. EVALUATION

This section presents the numerical results for validating the performance of the DDC against the SDC. Both the static scenario and dynamic scenario are considered.

A. Static Scenario

1) Parameter settings

The MILP can only find optimal solutions for small-size problems due to its high computational complexity. We consider only one pod consisting of one spine switch and 3 racks with one leaf switch, 3 blades per rack, and 3 components per blade. Three types of resources are considered: $R = \{CPU, memory, accelerator\}$, with a per component capacity of 100 units. The age of each hardware element is $U(10^9, 10^4)$ time units, where $U(a,b)$ is a random value uniformly distributed within the interval $(a,b)$. Henceforth, we do not add units to simulation time measures, which are understood to be time units. All the requests are assumed to be given at time 0, and the service time of each request is $U(1,100)$. Four kinds of requests are considered: CPU-intensive, memory-intensive, accelerator-intensive, and random. If request $i$ is of $r_1$-intensive ($r_1 \in R$), i.e., the first three kinds of requests, its demand is set as $D_{ir} = \{(U(40,80), r) = r_1 \mid U(10,40), r \neq r_1\}$. For a random request, its demand is $D_{ir} = U(10,80)$ $\forall r \in R$.

We consider four test cases considering the settings for reliability and network latency/bandwidth, namely, S1 – S4. In S1, all hardware elements except the resource components are resilient, and we assume that requests have no latency and bandwidth requirements. Each component’s reliability follows
the Weibull distribution with scale parameter $\eta_w = 10^6$ and shape parameter $\beta_w = U(1,2)$. This setting ensures that the hardware mean TBF (MTBF) is far longer than the request service time by four to five orders of magnitude. S2 is extended from S1, where blades become not resilient, whose reliability also follows the above Weibull distribution. Similarly, S3 is extended from S2, where each switch's reliability also follows the above Weibull distribution. Lastly, S4 further considers the latency constraints based on S3. With reference to the literature (e.g., [18], [40]), we set the latency-related parameters as follows. Hardware delays are set as: $\tau^{\text{bid}}_n = U(50,60)$ ns, $\tau^{\text{sw}}_n = U(100,150)$ ns, $\tau^{\text{bro}}_m = d \times L_{\text{pro}}$, where $L_{\text{pro}} = 5$ ns/m and $d = 2$m for each intra-rack link while $U(10,30)$ m for each inter-rack link. The latency requirement for the traffic between CPU and memory is $U(500,900)$ ns; $U(1,100)$ $\mu$s otherwise.

We use the commercial solver AMPL/Gurobi to solve the MILP formulations to provide solutions with respect to the objective function (3). The simulation environment for the heuristic algorithms is built through Java.

2) Results

![Fig. 9. Pareto fronts in terms of minimum request reliability vs. acceptance ratio for four different cases (S1 – S4) (obtained by MILP).](image)

We first evaluate the performance of different disaggregation scales using the MILP. Fig. 9 presents the results of the minimum requests reliability vs. the acceptance ratio in S1 – S4. Here, the number of requests is 30. In the figure, “BS”, “RS”, and “PS” correspond to the blade-scale, rack-scale, and pod-scale DDCs, respectively. We use dashed lines to connect the results to approximate the Pareto fronts [47]. Each approximate Pareto front provides a set of optimal solutions in terms of acceptance ratio and minimum request reliability based on given preferences. It is obtained by solving the MILP with varied weight factors [47]. The ordinate intervals in S1 – S4 are the same in Fig. 9 so that the differences between the Pareto fronts in S1-S4 can be clearly observed. We observe that PS performs the best in S1 and S2 while performing significantly worse in S3 and S4. This can be explained by the fact that PS has a larger disaggregation scale than RS and BS, so it can achieve higher minimum request reliability and acceptance ratio. However, a request in PS uses more network elements than in BS and RS. Therefore, when the switches are not resilient and latency requirements are considered, the performance of PS sharply decreases.

We also observe that RS performs better than BS in S1, while in S2 – S4, the performance of RS drops faster than BS. The reason is as follows. RS has higher flexibility and efficiency than BS, so RS outperforms BS in S1. However, service reliability in RS is significantly influenced by the network, so the performance of RS decreases sharply when blades and switches are not resilient (S2 – S3). In S4, the approximate Pareto fronts of RS and BS cross at (0.999955, 0.73). Accordingly, if setting a reliability threshold higher than 0.999955 (corresponding to the left side of the intersection), the acceptance ratio of RS is lower than BS while higher than BS otherwise. This is because requests in RS require more switches and blades than in BS, so their achievable reliability in RS may not be as high as in BS, and more requests in RS will be rejected if the reliability threshold is very high. On the contrary, when relaxing the threshold, the benefits of resource disaggregation manifest, where resource stranding in BS is significantly avoided in RS so that RS can achieve a higher acceptance ratio than BS. Overall, the performance difference of RS and BS on both sides of the intersection again demonstrates that resource disaggregation can improve resource efficiency, while the benefit will be offset if the network is not resilient.

![Fig. 10. Minimum request reliability vs. acceptance ratio obtained by Radar in S1 for BS and S3 for RS.](image)

Fig. 10 provides the results of Radar in S1 for BS and S3 for RS, and for comparison, we also show the Pareto fronts and other 10000 random solutions. The results of Radar are obtained with varied weighting coefficients in the blade and rack indices (See Section IV.B). We can observe that Radar performs close to the approximate Pareto fronts and outperforms all random solutions, demonstrating that Radar has high efficiency. In addition, Radar consistently maintains a high acceptance ratio. For example, in RS-S3, the acceptance ratio obtained by Radar is consistently larger than 0.8, also demonstrating its high efficiency. Note that in Fig. 10, we aim to show all the results obtained by Radar and Random, not only the optimal solutions. There are some solutions with the same acceptance ratio but different minimum request reliability obtained by Radar and Random, e.g., the two data points that have been circled in RS-S3. Clearly, having such two solutions, we will not choose the inferior one (the lower of the two). We only aim to illustrate such solutions and explain how they are obtained. This is explained through the 12th line of the algorithm in Fig. 4, which is the operation that sorts the blades in descending order of the blade index. Increasing the weighting coefficients of the blade index prioritizes highly reliable blades. However, the acceptance ratio may remain unchanged since the capacity does not change.
The above results demonstrate that an imperfect network will offset the benefits of hardware disaggregation. Backup is an intuitive way to improve reliability performance, and Fig. 11 gives the results for the case extended from S3, where every blade and switch has a backup. We observe from Fig. 11 that the results are very close to those in S1. This is reasonable as backup dramatically increases the reliability of switches and blades, making the performance very close to when the switches and blades are resilient. However, backup is not an efficient approach, and it leads to increased costs. Next, the results in the dynamic scenario show that improved performance is achievable without backup.

![Fig. 11. Pareto fronts for a new case extended from S3 by applying backup to every hardware element.](image)

**B. Dynamic Scenario**

1) **Parameter settings**

We consider one pod consisting of 10 spine switches and 9 racks with 2 leaf switches, 48 blades per rack, and 3 components per blade. Other settings, like the hardware capacity and request resource demand, are the same as in the static scenario. Here we do not consider the latency and bandwidth restrictions. We simulate a dynamic system, including the events of request arrival, request departure, hardware (components, blades, and switches) failure, and hardware repair. The request arrival follows a Poisson process with an arrival rate of 1.5, and service time follows an exponential distribution with a mean of 1000. The TBF of each element follows the Weibull distribution with parameters the same as in the static scenario. The time to repair follows another exponential distribution with a mean of 1. With this setting, hardware availability ranges from 99.999% to 99.9999%, which is consistent with hardware availability in real-world DCs [48]. In the simulation, we assume that the interrupted requests are immediately restored after the scheduler decides where to migrate the service. The simulation time is $10^6$.

2) **Results**

Fig. 12 gives the simulation results with varied weight coefficients. The first two subfigures give the results when migration is not applied. We observe that the blocking probability of PS and RS is significantly lower than that of BS. This is because of the significant efficiency benefit of hardware disaggregation. However, the second subfigure shows that, in terms of the number of accepted requests failing to complete service, PS has the highest value, second by RS and last by BS. The results further demonstrate that a vulnerable network may offset the benefits of disaggregation. The last two subfigures give the results when the migration is applied. We observe that PS and RS outperform BS in both blocking probability and the number of accepted requests failing to complete service. The results demonstrate that our proposed Radar framework can overcome the challenges created when the network is not resilient. Notice that in the static scenario, we have demonstrated that the performance of PS is significantly affected by latency. Therefore, only the RS can overcome the challenges caused by an imperfect network when considering latency.

![Fig. 12. Blocking probability and the number of requests failing to complete service with varying coefficients.](image)

**VIII. Conclusion and Future Research Directions**

We have studied service reliability in DDCs, considering network impact and different disaggregation scales. A MILP formulation and the Radar framework have been proposed for the resource allocation in DDCs, considering both reliability and utilization aspects. We have demonstrated that an imperfect network significantly offsets the improvement of resource efficiency and service reliability brought by hardware disaggregation. The results also demonstrate that hardware backup helps overcome such offset, which, however, may lead to increased cost. The proposed Radar framework employs a migration-based restoration, which can also overcome the offset without backup. Numerical results have shown that the rack-scale architecture is currently the best option, as a larger disaggregation scale faces latency challenges, which may lead to severe performance deterioration.

DDC is a relatively new research area that gives rise to many research opportunities. In the following, we consider the limitations of this work that lead to potential extensions and future research directions. Firstly, we have assumed that the demand required by a request is always additive from multiple resource components. This may not be the case in certain applications, e.g., a task/process should be executed on a single CPU component [12]. Secondly, the service request we have considered is like a VM or a task, which requires multiple different resources to form one VM or complete one task. However, there are some applications, such as parallel computing, where multiple VMs are required to jointly complete multiple tasks for a single job. Such applications have not been considered in this paper. Thirdly, as discussed, we have assumed no waiting room, and blocked requests are rejected without a re-allocation attempt. Finally, in response to
“the trends of AI for cloud, fog, edge, serverless, and quantum computing” to improve automation in DCs [49], we can study how to integrate AI tools like reinforcement learning with resource allocation in DCs.

REFERENCES
[41] Z. Ding, Y.-C. Tian, M. Tang, Y. Li, Y.-G. Wang, and C. Zhou, “Profile-guided three-phase virtual resource management for energy efficiency of


Chao Guo received his Master degree in information and communication engineering from Soochow University, Suzhou, China. He is currently working toward a Ph.D. degree with the Department of Electrical Engineering, City University of Hong Kong, Hong Kong. His research interests include the areas of data center resource scheduling and network optimization, network slicing, and cable path planning.

Moshe Zukerman (M’87–SM’91–F’07–LF’20) received the B.Sc. degree in industrial engineering and management, the M.Sc. degree in operations research from the Technion – Israel Institute of Technology, Haifa, Israel, and the Ph.D. degree in engineering from University of California, Los Angeles, in 1985. He was an independent consultant with the IRI Corporation and a Postdoctoral Fellow with the University of California, Los Angeles, in 1985–1986. In 1986–1997, he was with Telstra Research Laboratories (TRL), first as a Research Engineer and, in 1988–1997, as a Project Leader. He also taught and supervised graduate students at Monash University in 1990–2001. During 1997-2008, he was with The University of Melbourne, Victoria, Australia. In 2008 he joined City University of Hong Kong as a Chair Professor of Information Engineering, and a team leader. He has over 300 publications in scientific journals and conference proceedings. He has served on various editorial boards such as Computer Networks, *IEEE Communications Magazine, IEEE Journal of Selected Areas in Communications, IEEE/ACM Transactions on Networking and Computer Communications*.

Tianjiao Wang received a B.Sc. degree in electronic and communication engineering from Qufu Normal University, Shandong, China, and an M.Sc. degree in electronic and communication engineering from Beihang University, Beijing, China, in 2015 and 2018, respectively. She is currently working toward a Ph.D. degree with the Department of Electrical Engineering, City University of Hong Kong, Hong Kong. Her research interests include the areas of popularity prediction and cable network topology optimization.