Scheduling - A Secret Sauce For Resource Disaggregation

Alireza Farshin*
KTH Royal Institute of Technology
Stockholm, Sweden

Amir Roozbeh*
KTH Royal Institute of Technology
Stockholm, Sweden
Ericsson Research
Stockholm, Sweden

Christian Schulte
KTH Royal Institute of Technology
Stockholm, Sweden

Gerald Q. Maguire Jr.
KTH Royal Institute of Technology
Stockholm, Sweden

Dejan Kostić
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

This technical report describes the design & implementation of a constraint-based framework for scheduling & resource allocation in a disaggregated data center (DDC) where we build logical servers from disaggregated resources. We show that an Service Level Objective (SLO)-aware constraint-based solver could improve a data center's resource utilization by finding better solutions based on provided workload characteristics.

KEYWORDS

Disaggregated Data center, Scheduling, Resource Allocation, Gecode, Constraint Programming.

1 INTRODUCTION

Today's Data Center (DC) architecture follows the server-oriented model, where a DC is realized with pools of servers. These servers are made up of tightly integrated hardware (H/W) resources such as Central Processing Unit (CPU), memory, disks, and network interfaces. One of the key challenges of this architecture is low resource utilization [6, 22, 23, 28, 29, 38, 39], as a consequence of *resource stranding* (i.e., leftover and unused resources). Resource stranding occurs due to the mismatch between (fixed) servers' physical resources and applications' requirements. For instance, a server's compute resources could be exhausted by a CPU-intensive application while other resources, such as memory and storage, are still available.

To increase DC efficiency, DC providers usually employ advanced resource sharing techniques such as virtualization and containers. These techniques help DC providers to implement a multi-tenant cloud and perform efficient resource sharing & server consolidation, thereby improving DC resource utilization and reducing its costs. However, the benefits of these techniques are limited to servers' physical boundaries; therefore, DC resources still operate at low utilization [23, 29, 39]. To tackle this problem, the traditional server-oriented DC architecture is being challenged by a new architecture called H/W resource disaggregation [32, 33]. In this new architecture, the servers' boundary are broken and DC's resources become independent pools of different resource types. Hardware resource disaggregation provides a foundation that enables DC providers to perform fine-grained resource provisioning & scheduling thus more efficiently utilizing their deployed resources. A DC employing this architecture is called a Disaggregated Data Center (DDC). Fig. 1 illustrates the architecture of a server-oriented DC and a DDC.

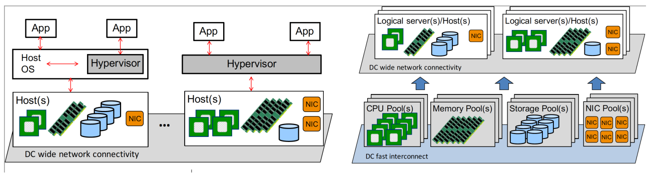


Figure 1: Server-oriented DC vs. Disaggregated DC. The left and right architectures show server-oriented and disaggregated models, respectively [32].

2 HARDWARE RESOURCE DISAGGREGATION

The idea of a disaggregated H/W architecture has captured the interest of both industry and research communities [32, 33]. In this architecture, DC resources are realized as resource pools that are interconnected via the DC's fast network. Resource pooling is expected to bring a high level of fluidity, modularity, and flexibility [42] to the DDC infrastructure. In addition, DDC allows the composition of *logical* servers by interconnecting physical resources from different resource pools. Hardware resource disaggregation provides a foundation that enables DC providers to perform fine-grained resource provisioning & scheduling, resulting in higher efficiency and utilization. In addition, DDC allows dynamically establishing and adapting the configuration of provisioned logical servers to precisely match the needs of particular workloads.

The transformation toward disaggregation of DC resources is expected to happen gradually [32]. Since there is no *de facto* standard for DDC, different communities have proposed somewhat different levels and types of resource disaggregation [16]. One example of these proposals is rack-scale design, which is a midpoint toward realizing datacenter-scale disaggregation [5, 11, 18, 19, 36]. In this case, the disaggregation is restricted to the rack level; hence, logical servers are built within a rack. However, this rack-scale design can be extended to include another rack equipped with only (or mainly) memory resources (*i.e.,* memory pool). This would result in a system with a multi-tier memory hierarchy, each of which can have different characteristics and costs.

Performance degradation is an inevitable part of H/W resource disaggregation due to the separation introduced among resources.

*Both authors contributed equally to the paper.

However, prior studies [2, 11, 14, 15, 21] have shown that this performance degradation could be bounded within a reasonable range under certain conditions. For instance, an application could achieve acceptable performance if it has access to at least a certain limited amount of local memory [43] (*e.g.,* sufficient for the working set in the case of virtual memory), which reduces the adverse impact of accessing data from a remote memory pool via the DC's fast network. Additionally, different resource pools may contain resources with different capabilities. For instance, two different memory pools can have different access latencies for reading/writing data (*e.g.,* Dynamic Random Access Memory (DRAM) vs. Non-Volatile Memory (NVM)). Therefore, one needs to consider all of these differences when composing the logical servers from the different resources to ensure that the resulting logical servers behave as an integrated system that meets the guaranteed Quality of Service (QoS) requirements, *i.e.,* those requirements defined in the Service-Level Agreement (SLA), aka Service Level Objective (SLO). For example, a customer could specify in the SLO that 95% percent of its requests should be processed/handled in less than 1 ms (*i.e.,* a defined SLO).

3 SCHEDULING AS THE SECRET SAUCE FOR BUILDING A DISAGGREGATED DC

Scheduling of workloads in a DC has received some attention in the literature in recent years, with surveys looking at energy aspects [40], load balancing techniques [37], Virtual Machine (VM) allocation & placement in the DC [10, 24], and others [2, 13, 17, 25, 26, 35, 41]. Overcoming the inherent problems of a DC built on disaggregated resources makes scheduling an essential factor for cloud providers. This scheduling can be realized at two levels:

(*i*) Scheduling & placement of data for the execution of a workload within these logical servers and scheduling different workloads in a composed logical server (aka job/task/data scheduling); and

(*ii*) Scheduling resources from different pools to build a logical server (aka resource scheduling/allocation).

Jobs/tasks scheduling. Jobs/tasks scheduling focuses on utilizing the allocated resources efficiently by executing different jobs/tasks based on their dependency, priority, and data availability/locality. Smart scheduling in a resource disaggregated DC creates opportunities to increase system performance. Given a one-to-one mapping between the workload and a logical host, the execution environment scheduler takes most of the responsibility for workload scheduling, including distribution of the workload's tasks and execution of each task's jobs. As a result, the Operating System (OS) and its corresponding execution environment can be modified to support scheduling that is aware of resource disaggregation and the details of remote vs. local resources (*e.g.,* memory) or even different resource types. As an example, the OS needs to be aware of the inter-connectivity characteristics (latency and bandwidth) of the composed logical server to make smart scheduling decisions. The OS needs to monitor (or be supplied with) the current state of various inter-connectivity characteristics in (near) real-time to make meaningful scheduling decisions. In this regard, Emmanuel Amaro, *et al.* [2] proposed a memory-aware scheduler that considers remote & local memory when assigning jobs and decides how to assign local memory to different jobs.

Resource scheduling/allocation. Resource scheduling/allocation involves selecting appropriate resources for a specific logical server based on their characteristics (*i.e.,* technological properties, topological information, and physical distance) to meet the required SLOs for a specific workload. A scheduling mechanism must take into account information about each individual resource (*e.g.,* CPU, memory, and storage) along with the DC's network (*i.e.,* link and switching fabric information). Scheduling mechanisms can be complex if the mechanism sees and selects every individual component or simpler when the mechanism only sees and selects the pool from which a certain resource should be allocated. One way to reduce the complexity of the core scheduling mechanism is to make each pool responsible for its internal scheduling, allowing individual optimization mechanisms per pool; thus, reducing the complexity of the core scheduling mechanism. Moreover, scheduling operations should not be static; hence, further optimization should be possible after allocations have been made. Some of the aspects that scheduling mechanisms should take into account are power consumption, resource defragmentation (*i.e.,* avoiding composing logical systems of resources that are too scattered), performance, and optimized utilization.

While the benefits of disaggregated environments have been extensively advocated, one needs to understand how to "carefully" foster them. When setting up a DC environment, dimensioning and distribution of resources come hand in hand. In a highly flexible environment with physical resource distribution, finding the optimal physical distribution & location of resources is even more important. For example, distributed resources bring an associated networking cost, making it necessary to find a balance between benefit and cost. Consider that separation of memory and CPU at long distances might be possible, but an expensive interconnect technology will eat into the potential utilization benefits. Bulent Abali, *et al.* [1] has assessed the cost of memory disaggregation. Moreover, another challenge is to understand and minimize the risk of correlated failures when placing CPU and memory pools in two separate chassis. For example, if a chassis hosting a CPU pool where several (logical) hosts are running, has a power failure, then all of these hosts will fail, *i.e.,* this physical co-location induces a correlation of failures in the logically independent servers. As a result, High Availability (HA) could be a constraint upon the resource composition mechanism. There are opportunities to realize different HA methods for a DDC other than those used in current server-centric DCs.

This technical report focuses on resource scheduling/allocation and proposes a constraint-based & SLO-aware resource allocation framework as a potential resource scheduling/allocation mechanism for a DDC. As noted earlier, the proposed framework can help cloud providers allocate resources in an efficient & economical way while meeting their customers' performance requirements (*i.e.,* SLOs). One can exploit this framework to find a suitable configuration of different resource pools for a given DC and the impact of SLA-aware resource allocation on a DCs's utilization will be discussed. To realize our goal, a real-world VM dataset [4] for resource allocation is used. The next section discusses the design and implementation of our framework. Our source code is publicly available at: aliireza/ddc-ra

4 SLA-AWARE FRAMEWORK FOR DDC

This section explains the models used in our proposed framework. First, we discuss a new way to map application-level characteristics (*e.g.,* service time) to a more tangible metric used to allocate resources (*e.g.,* memory) for logical servers within the DDC. Next, we explain the DC model based on resource disaggregation that is developed for the proposed framework. Finally, the section concludes by summarizing the constraint-based model used to allocate resources in the proposed framework.

4.1 SLA Model

As was discussed earlier, meeting SLOs can be an arduous task in a DDC, as resources allocated to a logical server might be physically separated, hence incurring higher latencies. Moreover, performance degradation occurs because of the increased separation between processors and memory. For instance, a processing core allocated to a logical server in a disaggregated DC might be located in a compute pool that is physically distant from the logical server's memory. Consequently, when the processor needs to access data, fetching the contents of the memory addresses from a remote physical location induces additional latency and this may result in the violation of SLOs. As noted earlier some applications can still achieve an acceptable response time if they have at least a certain limited amount of guaranteed local resources. Therefore, this work assumes that each compute pool will have some local memory [43]. Moreover, it is assumed that a disaggregated DC is deployed based on an extended version of the rack-scale design with multi-tier memory (*i.e.,* memory pools located in the same rack as the compute pool and memory pools located in other racks). Realizing an efficient DDC requires redefinition of SLA [20]. With this in mind, the high-level (application-level) SLOs are broken into two metrics, as follows:

- **SLO$_{\text{Local}}$** specifies the minimum amount of the local memory required for an application to be allocated from the local memory (M_{Local}) located within a compute pool:
$$M_{\text{Local}} \geq \text{SLO}_{\text{Local}} \quad (1)$$
- **SLO$_{\text{Remote}}$** represents the maximum weighted amount of remote memory which could be required by an application. In this case, we assume that the memory pool located in the same rack as the compute pool has a lower access cost. If a logical server's memory is allocated from multiple remote memory pools, we have:
$$\sum_i w_i \cdot M_{\text{Remote}_i} \leq \text{SLO}_{\text{Remote}} \quad (2)$$

where w_i denotes the cost of i^{th} remote memory pool (M_{Remote_i}). Section 4.2 elaborates on this.

Breaking/adapting SLO into these two components for a DDC is essential, as the traditional definitions of SLA/SLO were not designed for a disaggregated environment.

Extracting metrics. In this work, it is assumed that the allocation framework receives the secondary SLOs introduced above rather than the high-level SLO specified by the application. However, a real-world allocator for DDC could derive these auxiliary metrics from the application's memory access pattern, as different resource pools are expected to be shipped with a customized controller [36]. Additionally, application developers could extract these metrics from their applications. Some techniques that could help to derive these metrics (*i.e.,* local and remote SLOs) are:

I. **Profiling memory access pattern:** Application developers could profile & analyze the memory access pattern via existing tools (*e.g.,* Pin [30], Intel Processor Trace (PT) [31], and page fault statistics) and then calculate accurate or overestimated auxiliary SLOs for their applications.

II **Emulating disaggregated environment:** A DDC provider could provide its customers with an emulator that makes it possible for them to run their applications in different scenarios to directly calculate the secondary SLOs.

III. **Trial period:** A DDC provider could offer a trial period to its customers during which the required memory is *uniformly* allocated from all memory tiers. During this period the customer could measure the SLO violations and extract a more efficient memory allocation for later use.

IV. **Dynamic controller:** Alternatively, a DDC provider could measure the amount of SLO violations and then dynamically adapt the memory allocation at run-time.

This work assumes that the minimum amount of allocable memory is limited to 1 MB, but this could potentially be reduced to a smaller size (*e.g.,* 4-kB pages). Such a reduction could reduce memory fragmentation and thus result in fewer stranded resources in a disaggregated DC. Additionally, considering 4-kB pages as the minimum amount of allocable memory would be beneficial if the local memory of compute pools is expected to be used as an extension to Last Level Cache (LLC), which could be realized via page swapping [14].

4.2 Data Center Model

The framework assumes that a DDC is deployed based on an extended version of the rack-scale design, in which we have three memory tiers: (i) local memory located within a compute pool, (ii) a memory pool mounted in the same rack, and (iii) a memory pool situated in another rack. Additionally, the proposed model considers different costs for the different memory tiers (see Table 1), which will be used to meet the requirements of the remote SLO.

Table 1: Cost of different memory tiers in our framework.

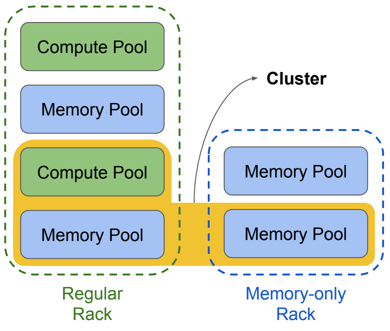
Memory Tier	Cost	Price ($/GB/hour)
Local (C_{Local})	0	100
Local remote ($C_{\text{LocalRemote}}$)	1	50
Distant remote ($C_{\text{DistantRemote}}$)	2	25

For instance, a logical server that requires 4 GB of memory (with local and remote SLOs equal to 1 GB and 4 GB, respectively) could only tolerate at most 1 GB of memory allocated from a distant remote memory pool according to Equations 3 and 4, where M and C stand for the memory size and the memory cost, respectively.

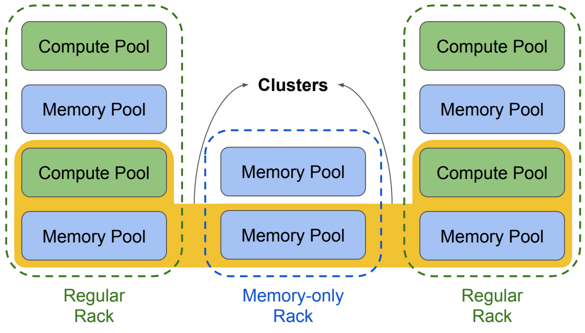
$$M_{\text{Total}} = M_{\text{Local}} + M_{\text{LocalRemote}} + M_{\text{DistantRemote}} \tag{3}$$

$$C_{\text{LocalRemote}} \cdot M_{\text{LocalRemote}} + C_{\text{DistantRemote}} \cdot M_{\text{DistantRemote}} \leq \text{SLO}_{\text{Remote}} \tag{4}$$

Pool clustering. In the proposed DDC model, a rack might contain multiple compute and memory pools, any combination of which could be considered during the allocation. However, considering all combinations could *dramatically* increase the time needed to find an optimal allocation. Therefore, the proposed framework breaks racks into smaller domains, called *clusters*, which each bundle a limited number of pools together, thereby limiting the search space during allocation. While introducing clusters imposes a new boundary to a DDC, this could be seen as a middle-ground solution that overcomes the server-oriented model in terms of increasing utilization. To push the boundary of this middle-ground solution and push its bounds, it is essential to choose a suitable model to cluster the different pools. This work considers two types of racks: (i) regular racks and (ii) memory-only racks. The former contains both compute pools and memory pools, while the latter only has memory pools. Fig. 2 shows two examples of pool clustering for our rack architecture. This work focuses solely on the first model (*i.e.,* Fig. 2a), where every cluster is composed of a compute pool, a memory pool, and a *nonshared* distant memory pool.



(a) Nonshared distant memory pools.



(b) Shared distant memory pools.

Figure 2: Two examples of pool clustering.

4.3 Constraint-based Model

The proposed framework uses a constraint-based solver, called *Gecode* [12]. The allocation problem is defined with the following constraints:

① *Allocating enough memory.* The proposed constrained-based resource allocator has to ensure that a logical server receives sufficient resources; therefore, we post a constraint in Gecode to satisfy Eq. 3.

② *Meeting SLOs.* To meet the SLOs, we post two additional constraints to guarantee the local & remote SLOs based on Equations 1 and 4.

③ *Minimizing the price.* The framework utilizes different prices for the different memory tiers (in addition to their cost). It is assumed that the price of memory increases as it gets closer to a compute pool, as shown in the third column of Table 1. The framework tries to minimize the deployment price of a logical server while meeting its SLOs. Therefore, a new constraint is posted on the price every time we find a new solution, which forces the next solution to have a lower price (as per Equations 5). It is worth mentioning that employing this strategy *implicitly* increases the resource utilization in a disaggregated DC, as it tries to use remote memory pools as much as possible.

$$\text{price} < \text{price}_{\text{current}} \tag{5}$$

Search strategy. The proposed allocator always starts allocating from the cluster with the maximum utilization to avoid turning on additional clusters. Additionally, it checks whether there are sufficient CPU cores available within a cluster for a given logical server *before* allocating memory. It is worth noting that at this stage the non-uniform architecture of the CPUs is not considered; however, Section 6 briefly elaborates the possibility of performing topology-aware core & CPU allocation.

First Fit (FF) allocator. To see the impact of meeting SLOs on the DC's utilization, the proposed constraint-based allocator is compared with a simpler heuristic allocator, called First Fit (FF), which does *not* consider SLOs during allocation. The FF allocator starts by allocating memories *sequentially* in different memory pools, *i.e.,* it first allocates from the local memory within the compute pool and then tries other memory tiers.

Implementation. The framework is written in C++ (~1800 lines of code). Fig. 3 provides a high-level overview of our design. The details of each module can be found in the source code at aliireza/ddc-ra.

5 EVALUATION

This section demonstrates the effectiveness of the proposed framework in different scenarios. It starts by describing the evaluation setup and then continues by presenting our results.

5.1 Setup

To evaluate the proposed framework, a real-world VM distribution (*i.e.,* frequency histogram) from Microsoft's Azure cloud is used to

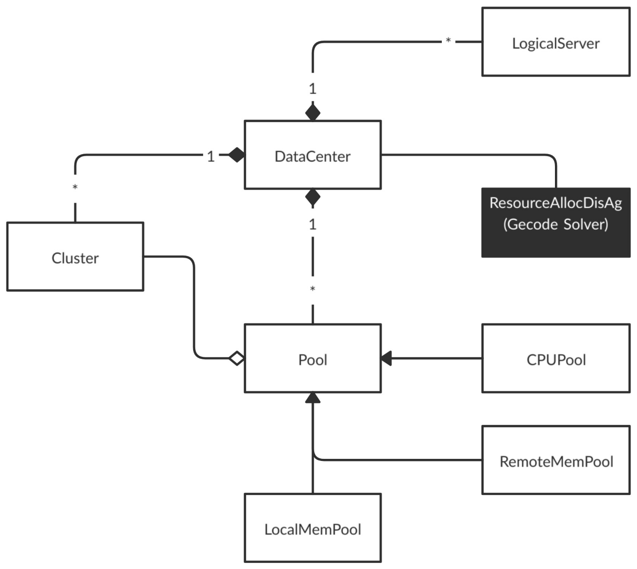


Figure 3: A high-level overview of our framework. Each box represents a C++ class.

generate 1000 logical servers with different numbers of CPU cores and different memory sizes[*] (see Table 2).

Table 2: Resource distribution of VMs in Microsoft Azure [4].

#Cores	**Frequency** (%)	**Memory**(GB)	**Frequency** (%)
2	58.2	2	12.1
4	22.9	4	16.5
8	12.0	8	32.8
12	0.1	32	28.7
24	5.6	64	6.4
30	1.2	70	3.6

It is assumed that every logical server requires 10% of its requested memory to be local (*i.e.,* $\mathrm{SLO}_{\mathrm{Local}}$). In addition, the $\mathrm{SLO}_{\mathrm{Remote}}$ is randomly generated for each VM based on the following equation, where α is a random number larger than 1:

$$\begin{aligned}\mathrm{SLO}_{\mathrm{Remote}} &= \alpha \times (M_{\mathrm{total}} - \mathrm{SLO}_{\mathrm{Local}}) \\ &= \alpha \times (M_{\mathrm{LocalRemote}} + M_{\mathrm{DistantRemote}})\end{aligned} \tag{6}$$

The value of α directly impacts the solvability of the problem due to the memory pools' maximum size and size. For instance, if we consider a logical server that requires 16 GB of remote memory with $\mathrm{SLO}_{\mathrm{Remote}}$ of 17 GB, then our allocator would be unable to allocate this logical server on a cluster (with costs similar to Table 1) when the size of a local remote memory pool is smaller than 16 GB. Therefore, we limit the value of α to be a random number in the range of $(1, 2)$. We also ensure that memory pools have sufficient space to accommodate the generated logical servers. Table 3 shows the configuration of the different pools in our evaluation setup.

Table 3: Pools' configurations. We assume that every compute pool contains 4 CPU sockets, each of which has 28 cores with 8 GB of local memory per socket. However, we do not consider Non-Uniform Memory Access (NUMA) in our framework. We specify 0 cores for memory pools, but they could have some available compute power to be used by their controller.

Type	**#Cores**	**Memory** (MB)
Compute pool	$4 \times 28 = 112$	4×8 GB = 32 GB
Local remote memory pool	0	128 GB
Distant remote memory pool	0	256 GB

5.2 Results

Initially, this section focuses on the *effectiveness* of our proposed allocator in terms of meeting the required SLOs and finding a solution with a minimal price. Later, it describes the impact of our allocator on the utilization of a DDC and its resources.

SLO violations. As discussed earlier, performance degradation is one of the main challenges in realizing a DDC. Therefore, choosing an appropriate resource scheduler/allocator becomes an essential factor. Fig 4 shows the percentage of violations for the FF allocator. These results demonstrate that using a *inappropriate* allocator (*e.g.,* FF allocator) could cause dramatic performance degradation. It is important to note that the proposed constraint-based allocator could always find a suitable allocation that meets the SLOs if there was any available solution.

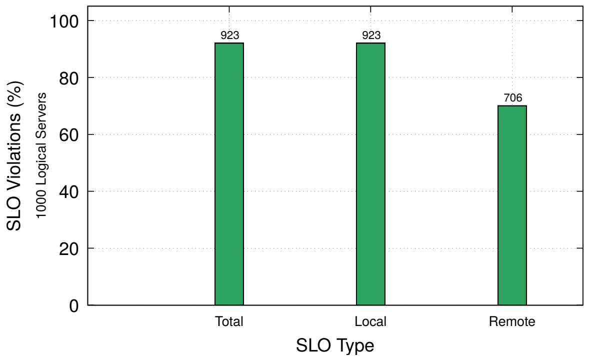


Figure 4: The percentage of logical servers (out of 1000) that could not meet their SLO, when allocated using a FF memory allocator. The number written above each bar shows the number of violations.

Price. One of the constraints considered in the proposed model is minimizing the price for a given logical server. Therefore, the cost of logical servers for both FF and the proposed constrained-based allocators are calculated and compared. Fig 5 shows the percentage

[*]The generated logical servers are available at `input.json`

of overpriced logical servers when we use the FF allocator as opposed to the proposed constrained-based framework. These results demonstrate that the proposed allocator could always find a cheaper allocation in cases where the SLO has been *met*. However, using the FF allocator could find cheaper allocations since it does not consider the SLOs — *i.e.,* due to the nature of FF allocation in which it could allocate all the required memory from the cheapest memory pool (*i.e.,* distant remote memory pool).

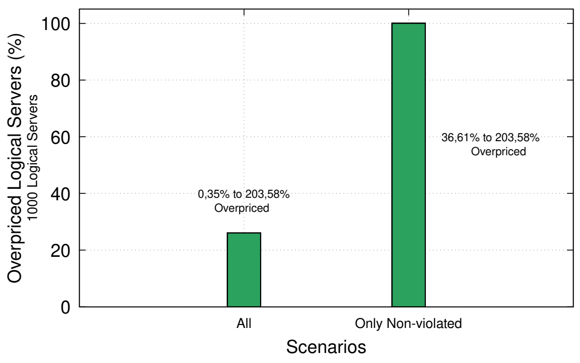


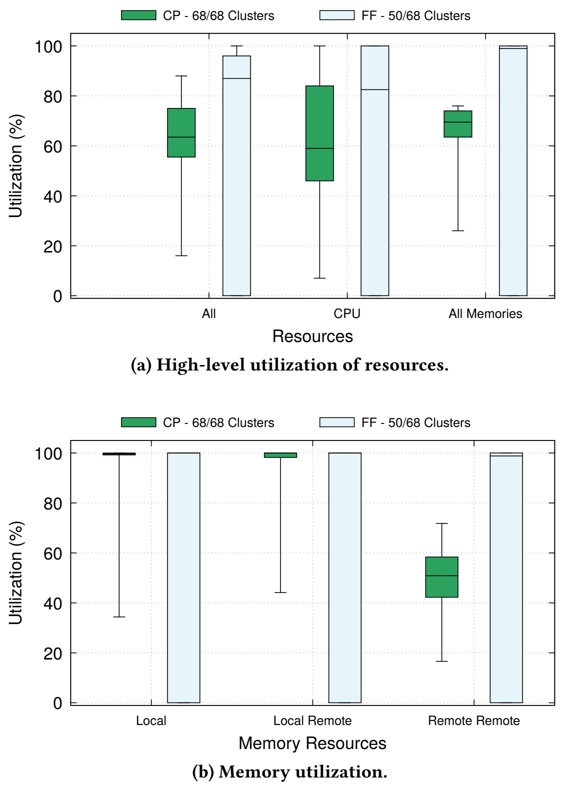
Figure 5: The percentage of overpriced logical servers (out of 1000) allocated using a FF memory allocator.

Utilization. The benefits of using a constrained-based resource allocator in a DDC to find a cheaper SLO-aware solution were discussed. Additionally, it is important to investigate the impact of considering SLOs while allocating resources on the utilization of a disaggregated DC. Fig. 6 compares the utilization of a DDC in both cases (*i.e.,* SLO-aware vs. non-SLO-aware). Fig. 6a shows that the FF allocator could always achieve better utilization. In addition, it could accommodate all of the logical servers in a smaller cluster (*i.e.,* 50 in FF vs. 68 in our framework). However, when comparing the memory utilization (see Fig. 6b), it is noticeable that the FF allocator *only* reaches a higher utilization for distant remote memory pools, which is mostly due to the chosen clustering model (see Fig. 2a). Since different clusters could not share a distant remote memory pool in the basic pool clustering model, this results in resource stranding if the other resources in a cluster are already occupied. However, other models that allow multiple (*e.g.,* two or more) clusters to share a distant remote memory pool could potentially achieve better resource utilization. Examining different clustering models remains as future work.

Execution time. The proposed constrained-based allocator takes much longer than the FF allocator (*i.e.,* 12.5804 s vs. 0.012 378 s) to allocate resource for 1000 logical servers in a DDC. However, the required time to find the solution with a minimal price is under 15 seconds, *i.e.,* quite reasonable.

6 CPU ALLOCATION

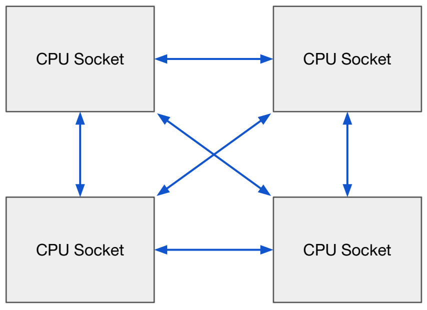
The previous sections only considered the availability of cores when allocating logical servers. As applications typically use multiple cores for parallelization, the inter-core communication of cores allocated to a VM/logical server can directly affect the applications' performance. For instance, real-world servers or CPU blades are often shipped with multiple processors, aka CPU sockets. The cost of communication between two cores residing on different CPU sockets is much more expensive than communication between cores located within a single CPU socket. Additionally, every processor interconnects multiple cores via a specific network-on-chip (NoC), which directly impacts the cost of inter-core communication. The impact of the communication is increasing since the number of cores is constantly increasing due to the demise of Dennard scaling [8]. Fig. 7 depicts an example of CPU-to-CPU and core-to-core interconnects. To achieve optimal performance, fine-grained CPU & core allocation should be performed and this should be based on the hardware's characteristics. Constraint programming is exploited to introduce a more optimized CPU allocator that considers the location of cores within a CPU socket. The proposed CPU allocator could be easily integrated into the framework introduced earlier. Furthermore, these same techniques could be extended to consider inter-socket communication and other important aspects required to achieve high-performance at high-speed networking [7] (*e.g.,* cache allocation [9]), which remains as future work.



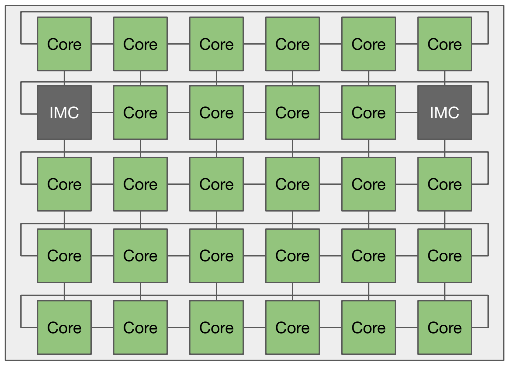
(a) High-level utilization of resources.

(b) Memory utilization.

Figure 6: Datacenter resource utilization when allocating logical servers using our allocator (*i.e.,* CP) and a first fit allocator (*i.e.,* FF). The FF allocator could allocate logical servers on 50 clusters, but we plot the distribution for 68 clusters, which is the minimum number of clusters required to meet the SLO objectives.



(a) CPU sockets topology.



(b) Cores topology.

Figure 7: An example of interconnections between CPUs and cores in Intel Xeon Scalable processors [27].

6.1 Model

To perform topology-aware core allocation, a CPU die with a two-dimensional torus interconnect is considered (*i.e.,* similar to Intel processors[*]). The core allocation problem is modeled as a tile covering problem, in which a set of tiles represent the cores. Consequently, the objective was to cover a torus topology with a set of tiles (C_i) while minimizing their pairwise distance. To solve this problem, *Gecode* [12, 34] was used to find the location $< X, Y >$ of tiles in a torus topology given a set of constraints described as follows:

① *Unoccupancy.* To assign a tile to a core, we have to ensure that the selected tile is free in the initial topology; therefore, if a tile is already occupied in the initial topology, it will be pruned from the search space. To do so, we restrict *at least one* of the coordinates of candidate tiles (*i.e., X* or *Y*) to be different from that of occupied tiles. If we show the occupied tile with *O*, we have:

$$\begin{aligned} \forall i, O : (X_{C_i} = X_O \wedge Y_{C_i} \neq Y_O) \quad &\vee \\ (X_{C_i} \neq X_O \wedge Y_{C_i} = Y_O) \quad &\vee \\ (X_{C_i} \neq X_O \wedge Y_{C_i} \neq Y_O) & \end{aligned} \tag{7}$$

[*]Although the details of routing in Intel processors are undocumented, we assume that it is using a typical XY routing algorithm [3]

② *Distinction.* To ensure the distinctiveness of the selected tiles, *non-overlapping* tiles should be picked. Consequently, *at least one* coordinates of every tile pair should differ.

$$\forall i, j, i \neq j : (X_{C_i} \neq X_{C_j}) \vee (Y_{C_i} \neq Y_{C_j}) \tag{8}$$

③ *Minimal distance.* The goal of the allocator is to minimize the distance of inter-core communication. Therefore, the allocator should select the tiles in such a way that their pairwise distance is minimal. To satisfy this criterion, a new constraint is posted so that when a solution is found, it restricts the distance of the next solution (*i.e., d*) to be smaller than the previous neighboring distance (*i.e.,* d_{current}).

$$d < d_{\text{current}} \tag{9}$$

The following formula is used to calculate the distance between two tiles, *i.e.,* d_{AB}, where A and B are two different tiles. n_{col} & n_{row} show the number of columns & rows in the CPU topology, respectively.

$$\begin{aligned} d_{AB} \quad = \quad &\min(n_{\text{col}} - |X_A - X_B|, |X_A - X_B|) + \\ &\min(n_{\text{row}} - |Y_A - Y_B|, |Y_A - Y_B|) \end{aligned} \tag{10}$$

$$d = \sum_{i,j} d_{C_i C_j} \tag{11}$$

6.2 Search strategy

Finding the optimal tile covering is known to be NP-hard. Therefore, it is essential to both reduce the search space size and perform the search efficiently. To do so, multiple optimizations are performed as follows:

Symmetry breaking. First, one of the symmetries in our core allocation problem is broken, *i.e.,* ignoring different permutations of the tiles ({<1,2>, <1,3>} vs. {<1,3>, <1,2>}). To do so, a constraint that forces the coordinates of the tiles to be in decreasing order is posted.

$$\forall i : (X_{C_i} \geq X_{C_{i+1}}) \vee (Y_{C_i} \geq Y_{C_{i+1}}) \tag{12}$$

Branching. Secondly, a problem-aware branching strategy is picked so that the optimal solution could be found faster during the search.

- As constraints have been defined to force decreasing order among coordinates; the variables are assigned from their maximum possible values.
- The X variables are chosen in a normal order (*i.e.,* first unassigned) while Y variables are selected based on their action parameter (*i.e.,* the number of active participation in the propagation).

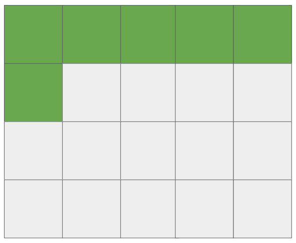
6.3 Evaluation

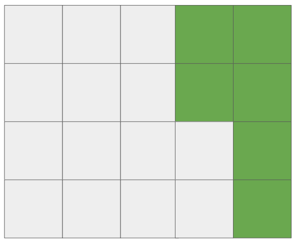
This section illustrates the effectiveness and discusses the scalability of the proposed CPU allocator.

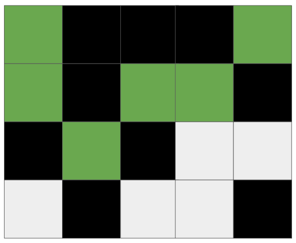
Allocation. To see the effectiveness of the proposed CPU allocator, the inter-core communication cost of the cores allocated via the proposed constrained-based allocator versus a FF core allocator[*] is compared. Fig 8 shows the results of the allocation for two different initial conditions where six cores should be allocated on a CPU with a 4×5 torus topology.

Empty CPU Figures 8a and 8b show the results for the case where the CPU is in its initial state, *i.e.,* no cores have been allocated yet. The cost of inter-core communication for FF and constraint-based allocation are 26 and 25, respectively.

Preoccupied CPU Figures 8c and 8d show the results for a case in which some of the cores were already allocated. The cost of inter-core communication for FF and constraint-based allocation are 34 and 29, respectively.


(a) CPU 1 - FF.


(b) CPU 1 - Our allocator.


(c) CPU 2 - FF.

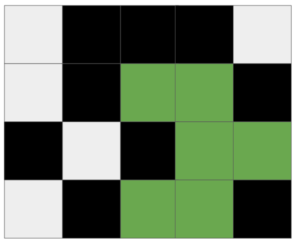

(d) CPU 2 - Our allocator.

Figure 8: An example of CPU allocation performed by our proposed allocator in a 4×5 torus topology. Black and green boxes show the pre-occupied and allocated cores, respectively.

By comparing the results in both cases, it can be realized that the proposed constrained-based allocator finds a better placement (*i.e.,* with smaller cost), as opposed to the FF allocator. However, the non-optimality of the solution found by the FF algorithm becomes more tangible in the second case, where some of the cores are pre-allocated (*i.e.,* 1 vs. 5).

Scalability. Table 4 shows the time for solving the problem with different topologies and a different number of cores. For the evaluation, scenarios with 2, 4, and 8 cores were considered, since these are typical requirements for VM allocation [4]. In addition, the evaluation focuses on topologies with a maximum of 49 cores, which is similar to the currently available CPUs with the maximum number of cores (*i.e.,* latest Cascade Lake AP chips with 48 cores). These results demonstrate that increasing the number of required cores can dramatically increase the solving time due to the search space explosion. More specifically, the optimal core placement in a 7×7 torus takes around ~40,000 seconds (*i.e.,* approximately 11 hours). Therefore, minimizing inter-core communication at run-time might not be possible, which is a reasonable result since tile covering is an NP-hard problem. Despite this limitation, performing topology-aware core/CPU allocation could still be useful for several reasons:

- The scalability evaluation focuses on the largest search space for a given topology, in which none of the cores are occupied. However, a real core allocator could use simpler heuristics/algorithms (*e.g.,* first fit) at first and then switch to a more advanced core allocator (*e.g.,* the proposed constrained-based framework) later. It was shown earlier that the non-optimality of simple heuristics is low when the CPU is not preoccupied. Additionally, when some of the cores have already been allocated the search space decreases.
- As optimal core allocation is mostly necessary for time-critical & high-performance applications; this feature could be provided on demand for those requests which are willing to wait longer for a solution.
- Core allocation could be performed using a more straightforward method at run-time and then be optimized periodically (*e.g.,* once a week) for the whole DC.

Table 4: Scalability of our proposed CPU allocator. The first and second columns (*i.e.,* "rows" and "columns") show the dimension of a torus topology for a CPU. The third column signifies the number of required cores for a hypothetical logical server and the fourth column demonstrates the time that it takes `Gecode` to solve the CPU allocation problem.

Rows	Columns	#Cores	Time (s)
4	4	2	0.0002
4	4	4	0.0179
4	4	8	350.6222
5	5	2	0.0002
5	5	4	0.0354
5	5	8	6956.7300
6	6	2	0.0002
6	6	4	0.0550
6	6	8	19795.4200
7	7	2	0.0002
7	7	4	0.0795
7	7	8	39298.7400

[*]FF allocates cores independently and starts from the first core (*i.e.,* top-left position in a torus topology)

7 CONCLUSION

This technical report demonstrated that an SLO-aware resource allocator could achieve an acceptable memory utilization if an appropriate clustering model is employed in DDC while meeting SLOs. It is essential to highlight that the initial resource distribution is performed based on workload forecasting. However, workload patterns likely change over time; hence, physical re-distribution of resources might be required. Therefore, having suitable mechanisms to optimize these re-distributions is necessary. The time required to find an appropriate allocation of resources via the proposed constrained-based scheduler might be unacceptable for some scenarios; therefore, evaluating other possible resource allocators/schedulers that can perform resource selection faster may be necessary, but remains as future work. However, the main objective of this report was to show that the scheduler could play a vital role in a DDC environment to make the logical system operate within acceptable SLOs while achieving high utilization and lower cost than not paying attention to the scheduling and allocation.

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work was also funded by the Swedish Foundation for Strategic Research (SSF). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

REFERENCES

[1] Bülent Abali, Richard J. Eickemeyer, Hubertus Franke, Chung-Sheng Li, and Marc Taubenblatt. 2015. Disaggregated and optically interconnected memory: when will it be cost effective? *CoRR* abs/1503.01416 (2015).

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[3] Shubhangi D Chawade, Mahendra A Gaikwad, and Rajendra M Patrikar. 2012. Review of XY Routing Algorithm for Network-on-Chip Architecture. *International Journal of Computer Applications* 975 (2012), 8887. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.734.7251&rep=rep1&type=pdf, accessed 2019-10-23.

[4] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)* (proceedings of the international symposium on operating systems principles (sosp) ed.). https://www.microsoft.com/en-us/research/publication/resource-central-understanding-predicting-workloads-improved-resource-management-large-cloud-platforms/

[5] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-scale Computers. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 551–564. https://doi.org/10.1145/2829988.2787492

[6] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 127–144. https://doi.org/10.1145/2654822.2541941

[7] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward per-Core 100-Gbps Networking *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3445814.3446724

[8] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[9] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 673–689. https://www.usenix.org/conference/atc20/presentation/farshin

[10] Alireza Farshin and Saeed Sharifian. 2019. A modified knowledge-based ant colony algorithm for virtual machine placement and simultaneous routing of NFV in distributed cloud architecture. *The Journal of Supercomputing* (16 March 2019). https://doi.org/10.1007/s11227-019-02804-x

[11] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Jo ao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[12] Gecode: GEneric COnstraint Development Environment. [n.d.]. https://www.gecode.org/, accessed 2019-10-17.

[13] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667.

[15] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network Support for Resource Disaggregation in Next-generation Datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (College Park, Maryland) *(HotNets-XII)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/2535771.2535778

[16] Tim Harris. 2015. Hardware Trends: Challenges and Opportunities in Distributed Computing. *SIGACT News* 46, 2 (June 2015), 89–95. https://doi.org/10.1145/2789149.2789165

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[18] Intel Corporation. 2015. *Intel Rack Scale Architecture using Intel Ethernet Multi-host Controller FM10000 Family.* Technical Report. Intel white paper. Online; accessed: 2017-06-21.

[19] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 690–695.

[20] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[21] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6168955

[22] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 450–462. https://doi.org/10.1145/2872887.2749475

[23] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. 2884–2892. https://doi.org/10.1109/BigData.2017.8258257

[24] Zoltán Ádám Mann. 2015. Allocation of Virtual Machines in Cloud Data Centers - A Survey of Problem Models and Optimization Algorithms. *ACM Comput. Surv.* 48, 1, Article 11 (Aug. 2015), 34 pages. https://doi.org/10.1145/2797211

[25] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[26] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot. 2019. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 15–27. https://doi.org/10.1109/HPCA.2019.00024

[27] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. Online; accessed 2018-09-22.

[28] M. Pawlish, A. S. Varde, and S. A. Robila. 2012. Analyzing utilization rates in data centers for optimizing energy management. In *2012 International Green Computing Conference (IGCC)*. 1–6. https://doi.org/10.1109/IGCC.2012.6322248

[29] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 183–190. https://doi.org/10.1109/SBAC-PAD49847.2020.00034

[30] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture* (Munich, Germany) *(WCAE '04)*. ACM, New York, NY, USA, Article 22. https://doi.org/10.1145/1275571.1275600

[31] James Reinders. 2013. Processor Tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, accessed 2019-10-29.

[32] Amir Roozbeh. 2019. Toward Next-generation Data Centers : Principles of Software-Defined Hardware Infrastructures and Resource Disaggregation. , 102 pages. Licentiat thesis, KTH.

[33] Amir Roozbeh, Jo ao Soares, Gerald Q. Maguire Jr., Fetahi Wuhib, Chakri Padala, Mozhgan Mahloo, Daniel Turull, Vinay Yadhav, and Dejan Kostić. 2018. Software-Defined Hardware Infrastructures: A Survey on Enabling Technologies and Open Research Directions. *IEEE Communications Surveys Tutorials* 20, 3 (May 2018), 2454–2485. https://doi.org/10.1109/COMST.2018.2834731

[34] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. 2019. Modeling and Programming with Gecode. (May 2019). https://www.gecode.org/doc-latest/MPG.pdf, accessed 2019-10-17.

[35] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[36] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[37] S. B. Shaw and A. K. Singh. 2014. A survey on scheduling and load balancing techniques in cloud computing environment. In *2014 International Conference on Computer and Communication Technology (ICCCT)*. 87–95. https://doi.org/10.1109/ICCCT.2014.7001474

[38] Junaid Shuja, Kashif Bilal, Sajjad Ahmad Madani, and Samee U. Khan. 2014. Data center energy efficient resource scheduling. *Cluster Computing* 17, 4 (2014), 1265–1277. https://doi.org/10.1007/s10586-014-0365-0

[39] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[40] Zoha Usmani and Shailendra Singh. 2016. A Survey of Virtual Machine Placement Techniques in a Cloud Data Center. *Procedia Computer Science* 78, Supplement C (2016), 491 – 498. https://doi.org/10.1016/j.procs.2016.02.093

[41] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[42] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. 2008. The Resource Pooling Principle. *SIGCOMM Comput. Commun. Rev.* 38, 5 (Sept. 2008), 47–52. https://doi.org/10.1145/1452335.1452342

[43] Yinghui Xing, Cen Wang, Ting Xu, Xiong Gao, Hongxiang Guo, and Jian Wu. 2020. Performance Evaluation of Distributed Computing over Optical Disaggregated Data Centers. In *2020 Asia Communications and Photonics Conference (ACP) and International Conference on Information Photonics and Optical Communications (IPOC)*. 1–3.