Subproject C4: On-The-Fly Compute Centers II: Execution of Composed Services in Configurable Compute Centers

Holger Karl[3], Marten Maack[2], Friedhelm Meyer auf der Heide[2], Simon Pukrop[2], Adrian Redder[1]

1 Department of Computer Science, Paderborn University, Paderborn, Germany
2 Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Paderborn, Germany
3 Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

1 Introduction

OTF compute centers are intended to exploit and support the characteristics of OTF services. They also are expected to exist at various scale, ranging from full-fledged data centers down to edge-computing semi-racks or even smaller. A characteristic of OTF services is that they are composed of components with explicit, quantitative meta data about those compositions – e.g., resource consumption per component, data flows between components, etc. OTF centers should therefore exploit this meta data to improve service performance and system efficiency. Moreover, OTF centers can be typically highly heterogeneous, having various types of computation units and persistent storage units. If a service provides metadata about its performance on different types of computation units, such information is also used to make better scheduling decisions as well. OTF centers also have one or more networks that connect these resources with each other. An OTF service can be provided by a single or several cooperating (sometimes also competing), geographically or organizationally distributed OTF compute centers. If necessary, they are supplemented by resources temporarily rented from the cloud.

The OTF services to be executed are usually composed of several interlinked, interacting components. Ideally, information about resource consumption, such as runtime and memory, is available for these components, possibly for different computing units such as CPUs, GPUs, and FPGAs. Information about the interaction of these components is available, such as the amount of data to be exchanged and minimum data rate requirements.

We have focused on resource management within as well as between data centers. We have worked from abstract, algorithmic models to very concrete framework-specific aspects, with methodological lines ranging from approximation and online algorithms with provable quality guarantees to system design and evaluation platforms. In doing so, we

holger.karl@hpi.de (Holger Karl), marten.maack@hni.upb.de (Marten Maack), fmadh@upb.de (Friedhelm Meyer auf der Heide), simonjp@hni.upb.de (Simon Pukrop), aredder@mail.upb.de (Adrian Redder)

have considered not only classical efficiency metrics such as throughput or utilization, but also energy consumption, for example.

In **Section 2.1:** *Approximation Algorithms for Scheduling*, we deal with the complexity of variants of scheduling problems. For this, we introduce new, and extend known models that capture important properties and features such as energy efficiency, the problems arising when global resources are available, the impact of setup times in reconfigurable systems, and the challenges arising when the compute center may delegate parts of the work to clouds. In these theoretical investigations, we have concentrated on algorithmic and complexity-theoretic approaches. On the one side, we have proven hardness results; on the other, we have developed approximation algorithms and proven bounds on their approximation quality.

In **Section 2.2:** *Distributed Execution of Service Chains*, we present extensions of formal description techniques towards OTF services. Further model extensions describe heterogeneous but interchangeable resources (e.g., CPU vs. FPGA). Based on these models, we have developed algorithms and mechanisms: which resources (data rate, compute capacity, ...) are allocated to which component to which task on which server. We have considered algorithms for both offline and online variants and have evaluated them both experimentally and theoretically. Hand in hand with the experimental analyses, we have also used the properties of real input streams (actual traces) as a starting point for modeling such streams in order to perform more realistic experimental analyses. In addition, we have developed heuristic or approximate solutions for these resource management problems and applied experiments and competitive analysis to evaluate their quality, partly based on realistic workloads.

2 Main Contributions

We structure the description of the main contributions of our subproject into the above-mentioned two sections.

2.1 Approximation Algorithms for Scheduling

The area of scheduling generally deals with the planned processing of tasks. From a computation perspective, addressing this topic leads to optimization problems that are typically combinatorial in nature and—in all but the simplest cases—NP-hard.[17] Hence, even if the complete instance of such a problem is known, there is little hope for an efficient algorithm that is guaranteed to find an optimal solution. One way of approaching this problem is to design algorithms that guarantee a certain quality in the produced solutions. In particular, an α-approximation for an optimization problem is guaranteed to produce a solution with an objective value that is within a factor of α of the optimum. The parameter α is called the *approximation ratio* or *guarantee* and, if not stated otherwise, the term *approximation algorithm* is used for algorithms that have a running time bounded by a polynomial in the input length of the problem. One of the earliest works in this direction was done by Graham [Gra66] in the 1960s regarding a fundamental scheduling problem.

[17] They cannot be solved efficiently if P ≠ NP, which is generally assumed to be true.

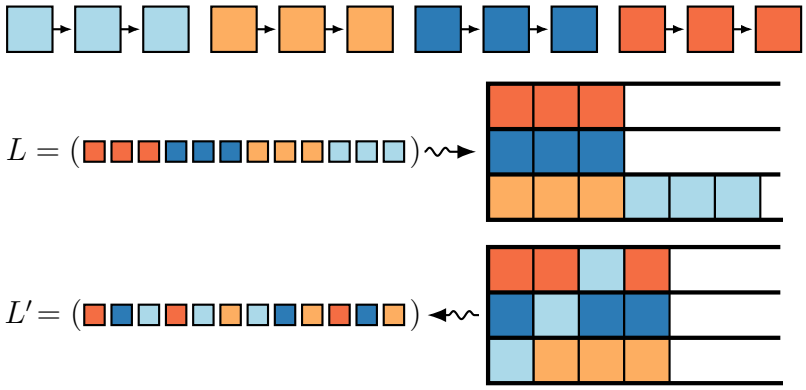


Figure 48: *A simple example for a scheduling problem with 12 unit time jobs with precedence constraints and three machines. For the first provided list L the list scheduling algorithm yields a schedule with makespan six while 4 is optimal. Generalizing this example, ratios arbitrarily close to 2 may occur. The list scheduling algorithm would have found an optimal schedule given list L′.*

We briefly discuss this classical result to make the topic more tangible and to introduce some of the basic concepts.

In the respective problem, a given set of jobs has to be assigned to a set of identical machines. Each job j has a processing time p_j, and between any pair of jobs (j, j') there may be a precedence constraint $j \prec j'$, that is, job j' can be processed only after job j is completed in this case. Furthermore, once the processing of a job is started, it cannot be interrupted (no preemptions), and the objective is to minimize the point in time in which the last job is completed—the *makespan*. Graham [Gra66] introduced the *list scheduling* algorithm for this problem, which arranges the jobs in a list and always schedules the first job on the list for which all precedence constraints are satisfied at the next possible time that is as soon as there is an idle machine. In this work, it was shown that list scheduling is a 2-approximation by considering two cases: Either all machines are working, or there are idle machines. In the first case, the algorithm behaves optimally, and in the second, either there are no more jobs or all the remaining jobs depend on the ones that are being executed. The latter observation can be used to bound the times with idle machines against the length of the longest chain of succeeding jobs which, in turn, is a lower bound for the optimum. Hence, both the times with and without idle machines can be upper bounded by the optimum, yielding the proof that list scheduling is a 2-approximation.[18] In Figure 48, an example is provided showing that the analysis cannot be substantially improved. Essentially, the best we can hope for regarding approximation algorithms for NP-hard problems are so-called approximation schemes: A *polynomial time approximation scheme (PTAS)* is a family of approximation algorithms (with polynomial running time) that provide a $(1 + \varepsilon)$-approximation for each $\varepsilon > 0$. Moreover, if the running time of the scheme is bounded by a polynomial in both the input length and $1/\varepsilon$, it is called *fully polynomial (FPTAS)*.

Interestingly, the list scheduling algorithm essentially still works in an *online* setting where the jobs are revealed over time during the processing time (they can be appended to the list). It is easy to see that in such an online algorithm, no algorithm can be guaranteed

[18] The actual analysis is slightly sharper.

to find an optimal solution for each input instance. However, there is an established way to measure the quality of an online algorithm that is closely related to the concept of approximation algorithms. In particular, it is considered c-competitive if the objective value achieved by the algorithm is guaranteed to be within a factor of c of an optimal *offline* solution.[19] Note that since the algorithm cannot know when the instance is completed, it has to *maintain* the above property for the respective instance seen so far. Coming back to the list scheduling algorithm as an example, it is easy to see that it barely uses any information about the instance, which is why the mentioned analysis can be adapted to show that it is 2-competitive.

Since the 1960s, the study of scheduling problems has expanded massively both in breadth and depth. For a broad overview, we refer to the textbook by Pinedo [Pin16]. In the context of the present subproject, however, approximation algorithms and to a much smaller extent, online algorithms have primarily been considered for areas of scheduling with particular relevance to OTF computing. In the following, we discuss the most prominent of the considered directions and highlight some of the most important results achieved in the subproject. Here we put the strongest focus on the topic we have dealt with the most at the end of the project, that is, cloud assisted scheduling.

2.1.1 Energy-Efficient Scheduling

In the study of scheduling, there is typically a strong focus on optimizing performance. Indeed, probably the most studied objective function in scheduling is the makespan, that is, the point in time the last task of an instance is completed. In many contexts, however, performance is neither the only nor the most important factor to consider, and one additional aspect that is of particular importance in today's world is energy efficiency. It is not quite obvious how to best capture energy consumption in a theoretical model, but a very influential approach to do so was introduced in 1995 by Yao et al. [YDS95] in a seminal work: In the *speed-scaling model*, the clock rates of processors can be changed at runtime, with slower clock rates resulting in lower overall energy consumption for the computation. A crucial factor is that the energy consumption grows superlinearly with the clock rate, with experiments pointing to growth with the third to fifth power in the clock rate for some real-world settings [BBS+00]. Moreover, the model relates to real-world techniques such as AMD's *PowerNow!* or Intel's *SpeedStep*.

As hinted above, a typical function modeling the energy consumption is of the form of s^α, where s is the clock speed of the processor, and α is a constant, usually between 3 and 5. In the classical work by Yao et al. [YDS95], a set of given jobs with different release dates, deadlines, and workloads has to be scheduled preemptively—the processing of a job may be interrupted, and resumed at a later time—on a single speed-scalable processor. The goal is to finish all the jobs in an energy-minimal way and both offline and online approaches are provided to that end. In a later work due to Chan et al. [CLL11], jobs are additionally associated with values and it is no more required to finish all jobs before their deadline but rather a combined objective of spend energy and lost profit is considered.

In [KP13], we generalize and improve upon the work by Chan et al. [CLL11]. We

[19] Depending on the problem, the concept is sometimes defined slightly differently, that is, up to additive constants.

consider the online setting and develop a combinatorial greedy algorithm that guarantees a competitive factor of α^α, which is optimal at least for greedy algorithms. In [CLL11], on the other hand, a $(\alpha^\alpha + 2e\alpha)$-competitive algorithm was presented for the single processor case. Moreover, the analysis of the algorithm uses techniques that are significantly different from the typical potential function argument. We utilize well-known tools from convex optimization and duality theory, in particular those that have already proved useful in the original work by Yao et al. [YDS95]. The developed algorithm, in some sense, can be seen as a combination of similar convex programming techniques with a carefully crafted greedy approach.

In [ABC+17], we consider a relaxed version of one of the central problems in speed-scaling: scheduling with respect to a combined objective of energy consumption and response time. While the problem regarding unit-sized jobs was well understood before, our results explore two important additional aspects, namely, arbitrary job sizes and discrete speed levels, which arguably model actual technology more accurately. Our results in [ABC+17] represent the first step in several years to solve the complexity question of this problem. More precisely, for the relaxation with *fractional* response times, we provide an efficient and optimal algorithm that follows a geometric approach utilizing certain structural properties that are obtained from an integer linear program and its dual.

2.1.2 Scheduling with Global Resources

In many real-world scheduling scenarios, different machines are connected via additional shared resources. Early considerations in this direction have already been made in the 1970s by Garey and Graham [GG75] building on the seminal work due to Graham [Gra66], discussed above. However, in the scheduling literature processors are very often assumed to be independent of each other. In contrast, we have considered models in which m identical machine share one additional resource, corresponding to, for instance, the data rate of a memory bus connecting processors being limited. The tasks to be processed are described by their processing times and resource requirement. The scheduler distributes tasks to processors and manages the access of the processors to the shared resource. If a task receives only a fraction of its resource requirement in a time step, its execution is slowed down accordingly. For example, a job of size p can be processed in p time units if it receives its full resource requirement in each time step. If it receives only half of its request in each time step, the processing time increases to $2p$. The goal is to minimize the makespan. Our key results regarding this scheduling problem are presented in [KMRS17]. We show the problem to be NP-hard and provide an efficient approximation algorithm with an approximation ratio of $2 + 1/(m-2)$. The algorithm utilizes a sliding windows approach that considers jobs ordered by non-decreasing resource requirement and, for each time step, tries to find a subset of consecutive jobs such that all but one can be completed using the full resource. Furthermore, we consider a variation of this model involving composed tasks consisting of multiple components, each of which has its own resource requirement. A task is completed when all of its components are completed and the goal is to minimize the average completion time of all task. We again show the problem to be NP-hard and provide an approximation algorithm with a ratio $2 + 4/(m-3)$ up to an additive constant.

2.1.3 Scheduling with Setup Times

In reconfigurable systems such as systems of FPGAs, a considerable amount of hardware configuration may be required when switching between tasks of different types. For scheduling in such systems, we have investigated a straightforward model in which we consider n tasks divided into k classes on a set of m processors. The processors have to be reconfigured to process the different classes. That is, each time a batch of jobs from a fixed class is to be processed on a processor a (possibly class dependent) setup time has to be paid. In [MMMR15], we provided the first results regarding this model with identical machines, including a $(3/2 + \varepsilon)$-approximation and an FPTAS for the case with a constant number of machines. This first result has quickly inspired further investigations from other researchers. For instance, Jansen and Land [JL16] provided a very simple and fast 3-approximation as well as a PTAS for the problem. In a follow-up work [JMM19], we considered generalized machine models: First, we developed a PTAS for the case with uniformly related machines, where the processing time of a job (and the setup time of a class) is scaled according to a machine-dependent speed factor. In the case of unrelated processing times (and setup times), we showed that no approximation algorithm with ratio $\Omega(\log n \log m)$ is possible unless a common hypothesis from complexity theory fails. We also provided a randomized algorithm with a matching upper bound. Lastly, we considered variants on identical machines with assignment restrictions and provided both hardness results and constant factor approximation algorithms.

2.1.4 Cloud Assisted Scheduling

Nowadays, a big part of web traffic and computational tasks are handled by large cloud providers such as Amazon Web Services and Microsoft Azure. Naturally concluding from that, a part of the CRC considered a setting in which computational resources are rented from cloud providers, exclusively or additionally. We present two different approaches, one where all jobs must be scheduled in the cloud and another where we own some free hardware that can be enhanced by rented cloud machines.

Cost-efficient scheduling on machines from the cloud: We consider the former approach in [MMMR18]. In that model, an online scheduler has to rent machines of a certain type for some arbitrary duration to ensure that all jobs can be scheduled before their respective deadline. Additionally, there is some machine-type dependent setup time s, before a newly rented machine can be used. The goal is naturally to minimize the cost paid to rent the machines. To be more specific, we assume that there are exactly two different machine-types, which differ in their price and their setup time. Jobs, on the other hand, consist of some processing time per machine-type, a release date, and a deadline. A critical parameter in this paper is the minimum slack β, which is the minimum time between any job release and the latest point where that job has to be scheduled to hit its deadline. Our paper has two main results: First, if the setups are large in comparison to the minimum slack ($s > \beta$) no finite competitiveness is possible. Secondly, if $\beta = (1 + \varepsilon)s$, for some ε with $1/s \leq \varepsilon \leq 1$, we give an algorithm that only depends on ε and the ratio of machine prices, and is proven to be optimal up to a factor of $O(1/\varepsilon^2)$.

Server cloud scheduling: In [MMP21] we both incorporate the possibility to allow some already owned hardware that can be augmented via the cloud, as well as imagining a big

task that can be represented as a graph of small jobs that depend on each other. This later part of our research combines various properties from different scheduling models, of which most have already been studied individually. Those are, in no particular order, unrelated machines, cost minimization for rentable machines, precedence constraints between jobs, and communication delays between the types of machines. We try to present this model in a bit more detail and describe one of the two main results.

We consider a scheduling problem SCS in which a task graph $G = (\mathcal{J}, E)$ has to be scheduled on a combination of a local machine (server) and a limitless number of remote machines (cloud). The task graph is a directed, acyclic graph. Each job $j \in \mathcal{J}$ has a processing time on the server $p_s(j)$ and on the cloud $p_c(j)$. The values of p_s and p_c can be arbitrary in $\mathbb{N}_0$, meaning that the server and the cloud are unrelated machines. An edge $e = (i, j)$ denotes precedence, i.e., job i has to be fully processed before job j can start. Furthermore, an edge $e = (i, j)$ has a communication delay of $c(i, j) \in \mathbb{N}_0$, which means that after job i finished, j has to wait for an additional $c(i, j)$ time steps before it can start, if i and j are not both scheduled on the same type of machine (server or cloud). A schedule π is a partition of the jobs into two sets: jobs processed on the server and the cloud, respectively. Additionally, a schedule assigns some starting time to each job. The cost (*cost*) of the schedule is then the total processing time of jobs processed on the cloud, and the makespan (*mspan*) is the completion time of the last job. For a schedule to be feasible, the following conditions must hold:

- Each job only starts after it is available, which means that all predecessors have finished processing and relevant communication delays have passed.
- No two jobs process on the server in parallel.
- If there is a budget, the *cost* may not exceed it.
- If there is a deadline, the *mspan* may not exceed it.

Naturally, if there is a budget, the goal is to minimize the deadline. If there is a deadline, the goal is to minimize the cost.

We categorize different sub-problems by their task graph structure and different processing times. The main results are an FPTAS with respect to the makespan objective for a fairly general case and strong hardness for the case with unit processing times and delays.

Imagine a task graph drawn in such a way that every edge goes from left to right. Now assume that we split the jobs in this task graph into a left part ($\mathcal{J}_l$) and a right part ($\mathcal{J}_r$), so that there are edges from $\mathcal{J}_l$ to $\mathcal{J}_r$, but no edges from $\mathcal{J}_r$ to $\mathcal{J}_l$. In other words, in a running schedule, $\mathcal{J}_l$ and $\mathcal{J}_r$ could represent already processed jobs and still be processed jobs, respectively. For any given task graph, we call the maximum number of edges between $\mathcal{J}_l$ and $\mathcal{J}_r$ the *maximum cardinality source and sink dividing cut* of the graph. We discuss how to solve or approximate SCS problems with a constant size cut, but otherwise arbitrary task graphs. We present the deadline-confined cost minimization; in the paper, we also show how to adapt this to the budget-confined makespan minimization. We start by describing a dynamic program to optimally solve instances of SCS with arbitrary task graphs. At first, we will not confine the algorithm to polynomial time. Consider a given problem instance with $G = (\mathcal{J}, E)$, processing times $p_s(j)$ and $p_c(j)$ for each $j \in \mathcal{J}$, communication delays $c(i, j)$ for each $(i, j) \in E$, and a deadline d. We define intermediate states of a (running) schedule as the states of our dynamic program. Such a state contains two types

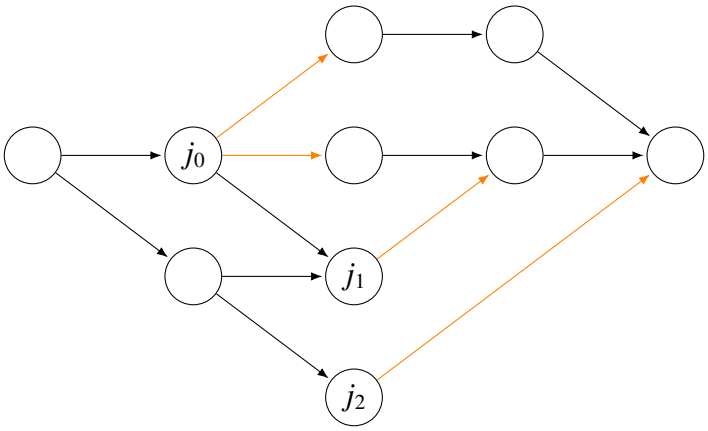


Figure 49: *Example for a small task graph with some state of a schedule.* j_0, j_1 *and* j_2 *represent jobs that are already processed but have some unprocessed successor remaining. Open edges are marked orange.*

of variables. First, we have two global variables, how many time steps have passed since the beginning and the number of consecutive time steps the server has been idling (counted from end to start). The second type is defined per *open edge*. An open edge is a $e = (j, k)$ where j has already been processed, but k has not. For each such edge $e = (j, k)$ add the following variables: the edge itself, whether j was processed on the server or the cloud, and the number of time steps passed that passed since j's completion. Note here that we purposefully drop the completion time and location of every processed job without open edges, as those are not important for future decisions anymore. There might be multiple ways to reach a specific state, but we only care about the minimum possible cost to achieve that state, which is the *value* of the state. We iteratively calculate the value of every state reachable in a given time step = $0, 1, 2, \ldots$. This state forms the beginning of our *state list*. We exhaustively calculate every state that is reachable during a specific time step, given the set of states reachable during the previous time step. Intuitively, we try every possible way to "fill up" the still undefined time windows of the server idling, and time passed since some j of an open edge was completed. After the current time step reaches our deadline, we can select the cheapest option from among the states to get the optimal schedule. This algorithm is polynomial in the deadline, but that can be exponential in the input size. To get an approximation algorithm that is polynomial in the input size, we scale all processing times in relation to the deadline and the input size. While doing so, we can $(1 + \varepsilon)$-approximate the optimal solution in time $poly(n, \varepsilon)$, for any $\varepsilon > 0$, which in turn means that we described an FPTAS for this problem. The other main result of the paper is a proof that the SCS problem is strongly NP-hard, even if all processing times and communication delays are equal to 1.

2.2 Distributed Execution of Service Chains

For the entire duration of the CRC, we have worked on basically the same scenario: the distributed execution of service chains in a complex infrastructural concept. Let us dissemble these terms first before digging into any more detailed contribution descriptions. First, the services we are considering are not monolithic services provided by a

single executable, e.g., a server process running somewhere. In line with developments in software engineering, a monolithic service executable is broken down into smaller independently executable pieces of software. They are connected together to collectively provide a service. The load in such a scenario can widely differ: A service may be invoked once by an individual user or repeatedly; a more interesting case is when a whole user population requests a service for repeated execution, an entire stream of requests arrives. The underlying infrastructure in such cases can be quite diverse: It can run from a tightly controlled data center to an edge-computing scenario in wide-area networks that is still under operational control of a single entity, to services and/or user populations that are spread over many administrative domains with independent control. In all these scenarios, there is a range of typical problems to solve in order to deal with load:

1. How many instances of a particular component service should be executed?
2. Where should these instances be placed?
3. Which request from which user is assigned to which instance; after processing one step in a service chain, requests from which instance are forwarded to which other instance?

These problems are known as the scaling, placement, and routing problems for service chains. In addition, there are further problems to solve, such as state management, deploying executable artefacts, etc. Most of the work described in this section deals with these problems under different perspectives.

In the following subsections, we first describe our contributions to these problems in the context of computing inside a wide-area operator network (In-network computing; Sections 2.2.1, 2.2.2, 2.2.3). Finally, in Section 2.2.4, we consider data center scenarios.

2.2.1 Description Techniques

When trying to deploy a service into a network, it is necessary to understand the characteristics and properties of such a service. From a purely functional perspective, it suffices to think of a service chain as a graph of atomic components, connected in a direct (typically, acyclic) graph with explicitly marked ingress and egress points. During operation, that knowledge suffices to forward one request along the chain (with additional information to which particular instance to forward to).

But during deployment (and reconfiguration), only functional information is insufficient to properly dimension resources. A better understanding of the required resources that a service or its components need is required. More specifically, what is the relationship between, on one hand (a) the load a component has to process (e.g., as a rate of temporal Poisson process) and (b) the resources that are assigned to it (e.g., the number of virtual cores, in a normalized manner) and, on the other hand, the resulting performance of such a component, e.g., the throughput it can sustain or the per-request delay. In early publications in this context, we have derived description formats to express such load-resource-performance profiles in a standardized manner, for individual components, services, and recursively defined services.

Here, we want to emphasize an aspect that has received little consideration and was first investigated in [SSKW19], jointly with colleagues from other CRC projects. The question

occurs what happens if the service graph splits into multiple paths from ingress to egress or if the service is request/reply-style, expecting an answer. Then, it matters whether subreplies from individual paths can be used independently or whether they need to be synchronized. Reference [SSKW19] shows that, all else being equal, this information matters for optimal deployment. That reference also introduced a Petri-net-based formalism to express such synchronization properties. Starting from a modeling formalism, we show that it is possible to automatically generate simulation programs or input files for Petri net solvers to assess the performance of concrete services. Moreover, thus modeled services can be fed into orchestration system that can leverage information about synchronization requirements for better orchestration decisions.

2.2.2 Orchestration

The above section has already mentioned the notion of "orchestration"—it is an umbrella term to capture all decisions that need to be taken when deploying a distributed service into a concrete infrastructure (e.g., as mentioned above, scaling, placement, and routing). In this section, we describe various algorithmic problems that we have talked in the orchestration context.

Conventional Orchestration Approaches A "conventional" approach is an approach that assumes full knowledge about the services to be orchestrated and their constituting performance profile, about the underlying infrastructure, and about the load patterns. This line of work culminated, in a sense, in Reference [DKM18]. The JASPER system proposed therein combined most of the aspects we had considered in previous papers and automatically deals with scaling, placement, and routing. It takes service templates and monitoring data of the underlying infrastructure as input and solves scaling, placement, and routing in an *integrated* optimization process (Figure 50), unlike separated, individual processes that were common in the literature before that. It handles dynamically adding services and services that terminate after completion, taking account of the current resource situation, and uses service templates in line with what was described before in Section 2.2.1. The reference also shows that the considered problem is NP-complete (via a set cover reduction proof). JASPER is also flexible in the way different optimization objectives can be combined and in that constraint violations can be acceptable, but their number is minimized. The solution approach is a mixed-integer linear program, with the typical limitations on problem size and require solution time. These limitations are amended by a heuristic. At the time of publication, a fairly unique feature of the heuristic was its capability to start from an existing solution and look for small modifications to accommodate new services upon arrival (Figure 51). This is considerably faster than always starting from scratch with marginal reductions of solution quality.

As a more specific example of optimization potential, we point out Reference [KK17]. It was one of the first papers to look at optimizing response time for such service chains, conceiving of the entire system as a queuing system where queuing delay is a dominant contributor to delay. The challenge was to find a good comprise for the non-linear time-in-system formula in a queuing system. We tackled this by developing a custom-tailored linear approximation to be used in a linear optimization program.

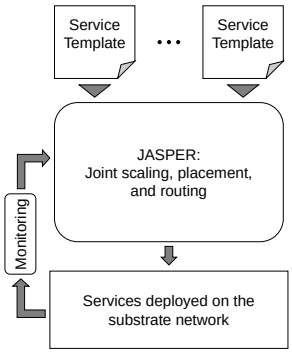
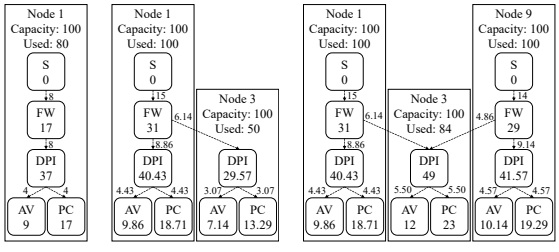


Figure 50: *Main control loop of the integrated JASPER approach for orchestration (based on Figure 2 in [DKM18]).*



(a) Initial embedding (b) Result of increased source data rate (c) Result of the emergence of a second source

Figure 51: *Steps of JASPER when load situation changes (Figure 6 in [DKM18]).*

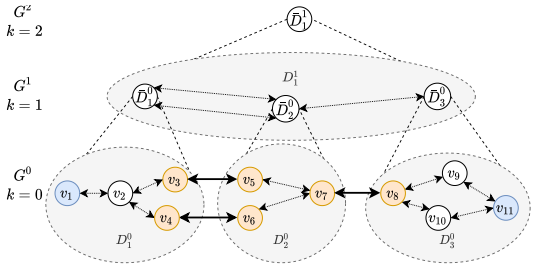


Figure 52: *Example with $k = 2$ hierarchies. Ingress and egress nodes are shown in blue, border nodes of a domain in orange (Figure 1 in [SJK21]).*

Hierarchical Orchestration Despite all improvements we did to the orchestration process, it still stayed a fairly complex problem. It stands to reason to break it down into subproblems. Moreover, in a multi-provider environment, it is unreasonable to assume that competing providers provide information about their infrastructure to each other. Both make a central perspective on the orchestration problem questionable. We hence developed a hierarchical approach to orchestrate services [SJK21] (Figure 52). The challenge was to find a good separation of available information and responsibility. Inspired by well-known multi-provider routing problems (known, e.g., from MPLS PCE contexts), we need to not only account for the data rates, but also for computational capacity. We did so by abstracting the capacity of a domain and only reported aggregated information to the higher level. Recursively, this ensured conservative orchestration choices trading off optimality for scalability.

Distributed Orchestration An alternative to hierarchical orchestration is to build an entirely distributed orchestration approach where each node works for itself. The challenge here is to deal with the non-locality of the orchestration problem: Resources might be available outside any node's observational horizon that still could result in a better solution. Hence, there is an inherent greediness involved in our distributed approach [SKK20].[20] More specifically, we looked at a locally greedy scheme—which processes

[20]It bears mentioning that this paper resulted from a Bachelor thesis.

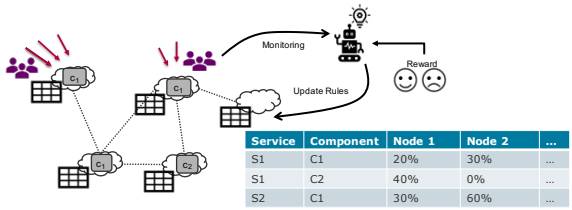


Figure 53: *Service orchestration as a centralized learning problem.*

any stage in a service chain once there is sufficient capacity available at a node—and combine it with an routing scheme where requests are forwarded on a shortest path towards their egress node, hoping that there will be sufficient capacity on the way to process remaining steps in a service chain (we did assume that every node can compute any stage in any chain, i.e., that all deployment units are available everywhere). This request forwarding does adjust to locally observable capacity information, rerouting a request away from already overloaded links. As a consequence, this scheme only needs global structural information (in particular, shortest paths) that change on long time scales and can reasonably be assumed to be available, but it does not assume non-local capacity or utilization information. It turns out that centralized heuristics are (unsurprisingly) still competitive with such distributed schemes but that distributed schemes achieve almost comparable performance at significantly reduced cost.

Machine-Learning-Based Orchestration All previously described orchestration approaches where "conventional" in the sense that they started from expert knowledge about the problem, the environment, and possible solution approaches. While this lead to interesting results and workable solutions, it is also promising to investigate currently popular machine-learning-based approaches and see how they fit to the orchestration problem. Specifically, reinforcement learning is a natural candidate, with an agent making orchestration decisions and obtaining rewards from the environment, e.g., informing it about how many flows could have been successfully processed or what relevant quality-of-service characteristics (e.g., request latency) were achieved. Typically, challenges to deal with are how to encode a network and services in fixed-length inputs and state representations necessary for an agent, how to encode suitable actions, and how to deal with delayed or sparse rewards or with uncertainty about service or infrastructure descriptions.

One way to deal with the state-size problem is investigated in Reference [SKM+21]. The key idea is, for each node, to use a table with service components as rows, other nodes as columns, and as an entry the probability with which to forward a request for a particular service component from one node to the node in the respective column. These tables are learned by a central agent based on delayed monitoring data and are periodically distributed back into the actual network. An illustratoin of the learning procedure is shown in Figure 53. The actual decisions are all taken locally (only needing a table lookup and a generation of a single random number). The results showed that such a centralized, delayed-observation-delayed-reward approach works surprisingly well, but it is obviously limited in scale. Separate tables need to be trained for each node.

To improve that situation, we compared it to another approach: Instead of training separate tables per node, we trained individual agents per node that had more freedom for decisions

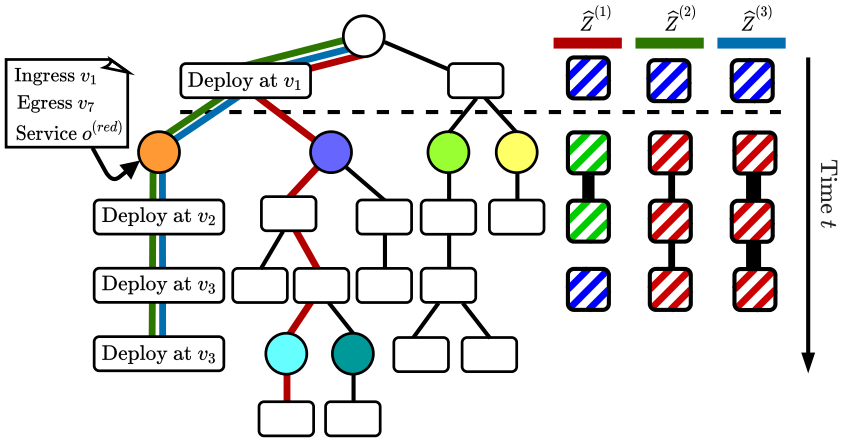


Figure 54: *FutureCoord plans service coordination beyond the current flow (left) with forecasts of future demands (right); Figure 2 of [WSK22].*

[SQK21]. This does improve the scalability of the overall concept in most cases; only in the case of a deadline for service execution is the first approach superior.

Approaches such as these are interesting, but they do entirely disregard any a priori understanding of the problem, at least at learning and inference time they are *model-free*. In addition, they do need expert knowledge to set up the representation of states, observations, and actions as well as to select the right reinforcement learning algorithms and neural network structures. It should make sense to incorporate explicit knowledge into a machine-learning approach, turning it into a *model-based* approach. FutureCoord [WSK22] is such an approach. Unlike many ML approaches, it is based on Monte-Carlo Tree search as the basic technique. It incorporates an explicit stochastic traffic model to use it for load predictions and to prepare the network for upcoming load changes (Figure 54). Basically, FutureCoord takes random samples from the stochastic traffic model, tries to optimize service orchestration along these samples, and picks a most promising action. As expected, the explicit inclusion of these traffic forecasts improves orchestration quality.

Distributed Machine Learning To make these ML approaches usable in a real system, we need to consider where and how to train these models. Transferring all data to a far-away cloud for training to later on retrieve the models is often not practical, given the amount of training data to transport and the frequency of training. Hence, we need to consider techniques for distributed, in-network machine learning. Notably, this problem is a problem in the context of networking for ML (or generally for computing). Here, the questions occurs what networking conditions need to be satisfied such that ML problems can be solved in distributed manner using a parallel computing infrastructure. In contrast, the distributed ML approaches discussed in the previous subsection considered the use of ML for networking, specifically network orchestration.

A key question in such distributed ML setups is whether and how fast a training algorithm converges. To accelerate the algorithm convergence, the space of learning variables is divided into several coordinates and multiple machines are assigned to work on each one asynchronously. The advantage of this approach is the potentially significantly enhanced convergence speed [ZZY+13]. However, the heterogeneity of computing resources and the

assignment of multiple machines to one coordinate induces that each machine effectively computes updates based on information with potentially significant age of information (AoI). The AoI arises because as one machine computes an update, all other machines may update their associated coordinate variables multiple times. It is therefore pertinent to address two problems: 1) When do modern machine learning algorithms work in the presence of the aforementioned AoI? 2) What is a representative model for AoI in asynchronous parallel computing architectures? In [RRK22b] we addressed the first problem. We developed AoI conditions for distributed stochastic gradient descent (DSGD), the main algorithm that underlies deep learning and artificial intelligence. Our conditions relate the cumulative distribution function of the AoI with the learning rates used by DSGD. The relationship shows that for highly parallelized architectures with many asynchronous machines (thus inducing large AoI), the GD updates should be performed with smaller learning rates to counter the error induced by the AoI. In [RRK22a] we addressed the second problem. We proposed a general model for AoI processes using event processes that possess dependency decay. For computing, our AoI model allows modeling of highly correlated traffic that share the parallel machines working on a machine learning problem. In summary, our two works therefore guide the choice of learning rates for machine learning algorithms running on parallel computing systems depending on the degree of correlation of arriving jobs.

Dealing with States When orchestrating services, an important distinction is whether the components are *stateless* or *stateful*: A stateless component obtains all required information to process a request from the request itself; a stateful component has to remember information from previous requests to process an actual one. Orchestrating stateless components is much simpler as there is no need to keep track of which request flow is mapped to which component, and rerouting can be done arbitrarily. But in reality, stateful components do appear.

One particular challenge is then to ensure that, when a flow is rerouted towards other components, the corresponding flow state is moved along. More specifically, we need to ensure proper timing. Once a request arrives at a new component, the flow state must have already been moved, but it must not be moved before the last request at the old component has not finished processing. It becomes necessary to synchronize flow and state migration with each other.

We tackled this problem by developing SHarP, a seamless handover protocol to integrate flow/state-migration protocol [PKK18b; PKK19] on top of an SDN-enabled network. The key idea is to use the SDN controller as a natural point of serialization by sending a handover message, a very limited number of request messages, and state handling messages via the controller (Figure 55 shows an intermediate step). This does impose additional load on the controller, but in experiments we were able to show that this overhead can be limited to a small number of messages, which should not create an unacceptable performance burden for SDN controllers.

2.2.3 Evaluation & Prototyping

A lot of the orchestration ideas described in the previous subsections were evaluated using simulations that use fairly simplistic models of the underlying system behavior—for

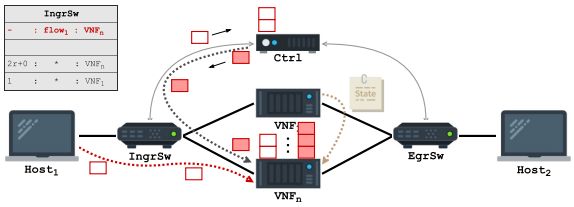


Figure 55: *An intermediate step in SHarP's state handover via ingress switch and SDN controller (Figure 3b of [PKK19]).*

example, the assumption that components of different services do not interact in their resource consumption and that components of the same service have natural dependencies, e.g., that the data rate sustainable by the slowest component determines the bottleneck data rate of an entire service. We were interested in double checking these assumption using actual experiments.

The challenge for such experiments is the required scale: For wide-area and data-center networks, we would need to run experiments on hundreds or thousands of nodes, which might only be feasible in rare circumstances and not amenable to continuous experimental work. Hence, we went for a compromise: to emulate actual environments, but run real code. To do so, we had to extend existing emulation tools. Starting from the well-known MiniNet tool, we first extended it to MaxiNet, enabling it to run in a distributed manner, scaling to thousands of emulated nodes. Then, we added the capability to run ordinary Docker containers as part of that emulation system, published as the tool ContainerNet [PKK18a] (with over 160 forks on GitHub as of late October 2022).

We used ContainerNet to construct a profiling platform for service components and entire services. It did turn out that it is necessary to profile services in their entirety [PK17] to properly reflect their internal interactions. To deal with all these aspects, a non-trivial system architecture emerged (Figure 56).

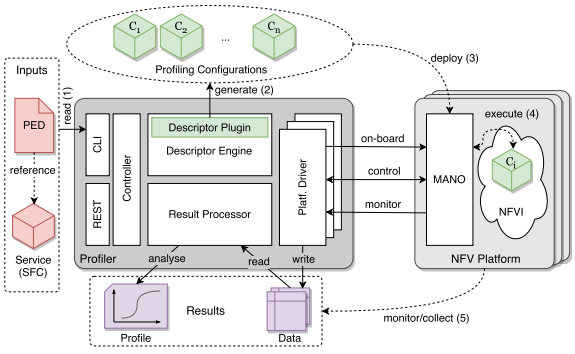


Figure 56: *System architecture of our profiling system interacting with several NFV platforms. The figure also shows the general workflow and generated artifacts (Figure 1 of [PK17]).*

2.2.4 Data Centers

In addition to service provisioning in wide-area networks, we also looked at data center scenarios. For example, we considered how to deal with so-called "coflows": a group

a flow that needs to be complete jointly before a distributed computation can continue its next stage (e.g., in a gather-collect context). We investigated machine-learning-based admission control and resource allocation schemes for that problem.

Here, we would like to describe an older contribution that addresses the following question: How can one generate traffic (e.g., for a simulation or emulation) that faithfully represents key statistical properties of actual data center traces? This is necessary as only limited amounts of traces are available, which is insufficient to drive performance evaluation work that is statistically meaningful.

In Reference [WK16], we describe a traffic generator that serves these needs; its main workflow is shown in Figure 57. Practically speaking, at the time of that work, traces on layer 2 (L2) were available, but for the scheduling work we were interested in, we needed layer 4 (L4) flow traces that were extensible yet faithful. To this end, we analyzed the available L2 traces and tried to infer L4 information from them. Checking whether this inference was correct is simple: Just use these L4 traces to run network experiments, collect L2 information, and compare statistical properties. The question is how to extract L4 information from L2 information, given that L2 information hides the bidirectional nature of TCP (packets are mirrored by acknowledgements in the opposite direction) and that this ACK traffic must not be mistaken for "actual" L4 traffic. We hence had to figure out the distribution functions for packet and ACK packet sizes and to "de-convolute" these different traces from the available L2 information. In the end, it turned out that we were able to construct corresponding L4 traces.

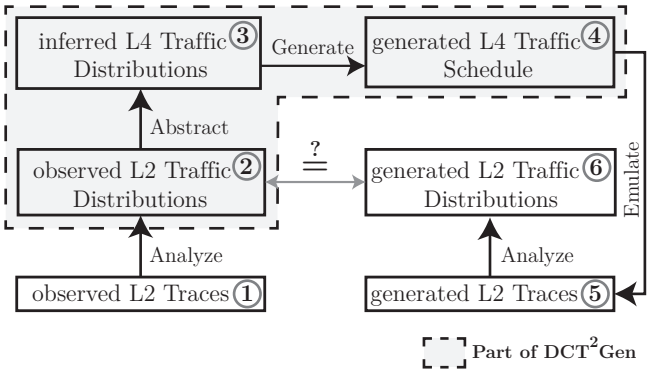


Figure 57: *Workflow of DCT2Gen (Figure 1 in [WK16]).*

3 Concluding Remarks

In Subproject C4, we considered the task of efficiently utilizing resources in highly configurable compute centers, be they big or small, centralized or distributed, from a variety of angles ranging from abstract algorithmic models to concrete framework-specific aspects. We conclude the discussion of these efforts by highlighting possible future research directions for some of the studied topics.

Scheduling with Setup Times

Setup times are extensively studied in the area of scheduling with a wide variety of different models. Regarding the class-based model considered in this subproject [MMMR15; JMM19] there are interesting open problems regarding closely related variants. For instance, the problem of identical machines with preemptions has been studied, i.e., where the processing of jobs may be interrupted and resumed at a later time. However, there is no PTAS known for this setting, and it seems challenging to design one. This is somewhat surprising given the fact that the scheduling problem without preemptions admits a PTAS, and the one with preemptions but without setup times is not even NP-hard. Moreover, considering the variant with preemptions for more general machine models would be interesting as well. Another interesting research direction can be derived from the fact that several novel PTAS results for scheduling with setup times have been obtained via newly developed techniques in the area of integer programming [JKMR22]. These techniques are based on utilizing some structure in the constraint matrix in order to derive provably efficient algorithms. It seems promising to further study the use of these techniques in the area of scheduling and to extend the techniques themselves to enable better or more general results, for instance, regarding scheduling with setup times. Lastly, there has been a recent trend to consider semi-online models in which crucial information regarding the instance is not known in advance, but estimates are given using a machine learning model, for instance. There have been several recent, intriguing results in this direction for scheduling problems presented at high-level conferences. It seems well worth considering scheduling with setup times—or other problems considered in this subproject—from this angle.

Cloud-Assisted Scheduling

As the Internet transforms into a landscape largely dominated by giant cloud service providers, cloud-assisted scheduling has become increasingly important. Since about 2010, there has been a plethora of different models, both practical and theoretical, that try to address some of the challenges that arise from this way of computing. A fundamental problem for theoretical analysis seems to be that there are so many different important properties of scheduling on clouds that a unifying model is currently out of reach. In no particular order, one might consider the leasing model and associated costs, job structure, precedence constraints, communication delays, release times, different machine speeds and capabilities, additional resources, online vs. offline algorithms, and more. Following on from this, it may be interesting to explore the limits at which generalizations of our model from [MMP21] no longer yield efficient (approximation) algorithms. In particular, the rental model and the cost function in our model are rather simple, we can get machines for exactly the time intervals we need, and we pay only in direct proportion to the jobs outsourced. A more elaborate and realistic leasing system for cloud resources, including other costs and start-up times for new machines, could provide interesting insights. Finally, in the context of the CRC, we would like to mention that this issue can also be explored from a market perspective itself, where multiple cloud providers compete to schedule customers' jobs efficiently in order to maximize their own profits.

In-Network Computing

Since the start of this CRC, the notion of in-network computing has changed substantially. By now, it is fairly commonplace to find discussions about many different forms of infrastructure, spreading the resources of data centers ever more thinly across real environments. A common buzzword in this context is the "edge-cloud continuum," where along the path from a device to a centralized cloud, many different forms of service execution opportunities exist, e.g., gateways or micro-cloud data centers of various forms. Somethings, about a dozen or so different stages are differentiated. Often, it is not entirely clear what the differences are, but architectures exist that go to great pains to make such differences and assign different roles, APIs, etc. We believe that artificially introducing differentiation where none exists is detrimental to both the efficiency and uptake of such concepts. We argue that consistent, simple concepts to distribute composed services are to be much preferred, but they have not really materialized, despite a lot of practical progress in using resources of different cloud providers. There is no consistent approach in sight.

We do acknowledge, however, that there are differences in business models associated with such a multi-stage infrastructure. While that certainly drives competition and can be a strong hindrance to standardization towards common APIs, there are also actual consequences. For example, there is no clear notion of a "chain of custody" for storing data or executing services. While fundamentally, this is in many forms unsolvable (essentially, the impossibility of consensus in faulty, asynchronous systems), there still is a need for practical compromises with a clear assignment of responsibilities and custody for data or services. Again, while there is a lot of understanding available about basic concepts and their limitations, there is no agreed standard that would foster the adoption of such architectures.

Prototyping and Actual Experiences

Very much in the same vain, we believe that there is a need for more experimental experience in real systems. A lot of results come from simulation or carefully controlled lab environments and emulations, but there is not much academic work done in real environments "in the wild." As of today, this is still the purview of cloud providers, hyperscalers and major over-the-top providers such as Netflix. This is a particular problem for most work that follows machine-learning approaches: When no data is available, there is nothing from which a model can be learned, and there is even less opportunity to really test approaches to manage data centers (at whatever stage of a continuum). We believe that this methodical gap needs to be closed, but there is no obvious approach how to do that. This is a challenging area for systems research the coming years, which ensures that our work stays relevant by being able to work from and test in relevant environments using relevant data.

Bibliography

[ABC+17] Antoniadis, A.; Barcelo, N.; Consuegra, M. E.; Kling, P.; Nugent, M.; Pruhs, K.; Scquizzato, M.: Efficient Computation of Optimal Energy and Fractional Weighted Flow Trade-Off Schedules. In: vol. 79. 2. 2017, pp. 568–597.

[BBS+00] Brooks, D. M.; Bose, P.; Schuster, S.; Jacobson, H. M.; Kudva, P.; Buyuktosunoglu, A.; Wellman, J.; Zyuban, V. V.; Gupta, M.; Cook, P. W.: Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. In: *IEEE Micro* 20 (2000), no. 6, pp. 26–44.

[CLL11] Chan, H.; Lam, T. W.; Li, R.: Tradeoff between energy and throughput for online deadline scheduling. In: *Sustain. Comput. Informatics Syst.* 1 (2011), no. 3, pp. 189–195.

[DKM18] Dräxler, S.; Karl, H.; Mann, Z. A.: JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services. In: *IEEE Transactions on Network and Service Management* (2018)

[GG75] Garey, M. R.; Graham, R. L.: Bounds for Multiprocessor Scheduling with Resource Constraints. In: *SIAM J. Comput.* 4 (1975), no. 2, pp. 187–200.

[Gra66] Graham, R. L.: Bounds for certain multiprocessing anomalies. In: *Bell system technical journal* 45 (1966), no. 9, pp. 1563–1581

[JKMR22] Jansen, K.; Klein, K.; Maack, M.; Rau, M.: Empowering the configuration-IP: new PTAS results for scheduling with setup times. In: *Math. Program.* 195 (2022), no. 1, pp. 367–401.

[JL16] Jansen, K.; Land, F.: Non-preemptive Scheduling with Setup Times: A PTAS. In: *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. Ed. by Dutot, P.; Trystram, D. Vol. 9833. Lecture Notes in Computer Science. Springer, 2016, pp. 159–170.

[JMM19] Jansen, K.; Maack, M.; Mäcker, A.: Scheduling on (Un-)Related Machines with Setup Times. In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 145–154.

[KK17] Keller, M.; Karl, H.: Response-Time-Optimised Service Deployment: MILP Formulations of Piece-wise Linear Functions Approximating Non-linear Bivariate Mixed-integer Functions. In: *IEEE Transactions on Network and Service Management* (2017), no. 1, pp. 121–135

[KMRS17] Kling, P.; Mäcker, A.; Riechers, S.; Skopalik, A.: Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*. Ed. by Scheideler, C.; Hajiaghayi, M. T. ACM, 2017, pp. 123–132.

[KP13] Kling, P.; Pietrzyk, P.: Profitable scheduling on multiple speed-scalable processors. In: *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*. Ed. by Blelloch, G. E.; Vöcking, B. ACM, 2013, pp. 251–260.

[MMMR15] Mäcker, A.; Malatyali, M.; Meyer auf der Heide, F.; Riechers, S.: Non-preemptive Scheduling on Machines with Setup Times. In: *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*. Ed. by Dehne, F.; Sack, J.; Stege, U. Vol. 9214. Lecture Notes in Computer Science. Springer, 2015, pp. 542–553.

[MMMR18] Mäcker, A.; Malatyali, M.; Meyer auf der Heide, F.; Riechers, S.: Cost-efficient scheduling on machines from the cloud. In: *J. Comb. Optim.* 36 (2018), no. 4, pp. 1168–1194.

[MMP21] Maack, M.; Meyer auf der Heide, F.; Pukrop, S.: "Server Cloud Scheduling." In: *Approximation and Online Algorithms - 19th International Workshop, WAOA 2021, Lisbon, Portugal, September 6-10, 2021, Revised Selected Papers*. Ed. by Könemann, J.; Peis, B. Vol. 12982. Lecture Notes in Computer Science. Springer, 2021, pp. 144–164.

[Pin16] Pinedo, M. L.: *Scheduling: Theory, Algorithms, and Systems*. Springer, 2016

[PK17] Peuster, M.; Karl, H.: Profile Your Chains, Not Functions. Automated Network Service Profiling in DevOps Environments. In: *IEEE Conference on Network Function Virtualisation and Software Defined Networks (NFV-SDN)*. Berlin, 2017

[PKK18a] Peuster, M.; Kampmeyer, J.; Karl, H.: Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains. In: *4th IEEE International Conference on Network Softwarization (NetSoft 2018)*. Montreal, 2018

[PKK18b] Peuster, M.; Küttner, H.; Karl, H.: Let the state follow its flows: An SDN-based flow handover protocol to support state migration. In: *4th IEEE International Conference on Network Softwarization (NetSoft 2018)*. Montreal, 2018

[PKK19] Peuster, M.; Küttner, H.; Karl, H.: A flow handover protocol to support state migration in softwarized networks. In: *International Journal of Network Management* (2019)

[RRK22a] Redder, A.; Ramaswamy, A.; Karl, H.: Age of Information Process under Strongly Mixing Communication – Moment Bound, Mixing Rate and Strong Law. In: *Proceedings of the 58th Allerton Conference on Communication, Control, and Computing*. 2022

[RRK22b] Redder, A.; Ramaswamy, A.; Karl, H.: Practical Network Conditions for the Convergence of Distributed Optimization. In: *IFAC-PapersOnLine* 55 (2022), no. 13, pp. 133–138

[SJK21] Schneider, S. B.; Jürgens, M.; Karl, H.: Divide and Conquer: Hierarchical Network and Service Coordination. In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. Bordeaux, France: IFIP/IEEE, 2021

[SKK20] Schneider, S. B.; Klenner, L. D.; Karl, H.: Every Node for Itself: Fully Distributed Service Coordination. In: *IEEE International Conference on Network and Service Management (CNSM)*. IEEE, 2020

[SKM+21] Schneider, S. B.; Khalili, R.; Manzoor, A.; Qarawlus, H.; Schellenberg, R.; Karl, H.; Hecker, A.: Self-Learning Multi-Objective Service Coordination Using Deep Reinforcement Learning. In: *Transactions on Network and Service Management* (2021)

[SQK21] Schneider, S. B.; Qarawlus, H.; Karl, H.: Distributed Online Service Coordination Using Deep Reinforcement Learning. In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. Washington, DC, USA: IEEE, 2021

[SSKW19] Schneider, S. B.; Sharma, A.; Karl, H.; Wehrheim, H.: Specifying and Analyzing Virtual Network Services Using Queuing Petri Nets. In: *2019 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. Washington, DC, USA: IFIP, 2019, pp. 116–124

[WK16] Wette, P.; Karl, H.: DCT²Gen: A traffic generator for data centers. In: *Computer Communications* (2016), pp. 45–58

[WSK22] Werner, S.; Schneider, S. B.; Karl, H.: Use What You Know: Network and Service Coordination Beyond Certainty. In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. Budapest: IEEE, 2022

[YDS95] Yao, F. F.; Demers, A. J.; Shenker, S.: A Scheduling Model for Reduced CPU Energy. In: *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*. IEEE Computer Society, 1995, pp. 374–382.

[ZZY+13] Zhang, S.; Zhang, C.; You, Z.; Zheng, R.; Xu, B.: Asynchronous stochastic gradient descent for DNN training. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 6660–6663