On Energy Conservation in Data Centers

Extended Abstract

Susanne Albers*
Technical University of Munich
85748 Garching, Germany
albers@in.tum.de

ABSTRACT

We formulate and study an optimization problem that arises in the energy management of data centers and, more generally, multiprocessor environments. Data centers host a large number of heterogeneous servers. Each server has an active state and several standby/sleep states with individual power consumption rates. The demand for computing capacity varies over time. Idle servers may be transitioned to low-power modes so as to rightsize the pool of active servers. The goal is to find a state transition schedule for the servers that minimizes the total energy consumed. On a small scale the same problem arises in multi-core architectures with heterogeneous processors on a chip. One has to determine active and idle periods for the cores so as to guarantee a certain service and minimize the consumed energy.

For this power/capacity management problem, we develop two main results. We use the terminology of the data center setting. First, we investigate the scenario that each server has two states, i.e. an active state and a sleep state. We show that an optimal solution, minimizing energy consumption, can be computed in polynomial time by a combinatorial algorithm. The algorithm resorts to a single-commodity min-cost flow computation. Second, we study the general scenario that each server has an active state and multiple standby/sleep states. We devise a τ-approximation algorithm that relies on a two-commodity min-cost flow computation. Here τ is the number of different server types. A data center has a large collection of machines but only a relatively small number of different server architectures. Moreover, in the optimization one can assign servers with comparable energy consumption to the same class. Technically, both of our algorithms involve non-trivial flow modification procedures. In particular, given a fractional two-commodity flow, our algorithm executes advanced rounding and flow packing routines.

KEYWORDS

Heterogeneous machines; efficient algorithms; approximation algorithms; minimum-cost flow.

*Work supported by the European Research Council, Grant Agreement No. 691672.

SPAA '17, July 24–26, 2017, Washington, DC, USA
© 2017 Copyright held by the owner/author(s). 978-1-4503-4593-4/17/07.
DOI: http://dx.doi.org/10.1145/3087556.3087560

1 INTRODUCTION

We define and investigate an optimization problem with the objective of energy conservation in multiprocessor environments. We focus on two particularly timely settings.

Data centers. Energy management is a key issue in data center operations [7]. Electricity costs are a dominant and rapidly growing expense in such centers; about 30-50% of their budget is invested into energy. Data centers use about 1.5% of the total electricity worldwide [12]. This corresponds to the energy consumption of more than 90 million households [8]. Surprisingly, the servers of a data center are only utilized 20–40% of the time on average [3, 6]. When idle and in active mode, they consume about half of their peak power. Hence a fruitful approach for energy conservation and capacity management is to transition idle servers into standby and sleep states. Servers have a number of low-power states [1]. However state transitions, and in particular power-up operations, incur energy/cost. Therefore, dynamically matching the varying demand for computing capacity with the number of active servers is a challenging problem.

Multi-core architectures. Multi-core processors are architectures with multiple, often heterogeneous processing units on a single die. Originally, heterogeneous platforms contained several processor types, i.e. CPUs and GPUs. Modern platforms are also equipped with identical CPUs that have different micro-architectures leading to various levels of energy consumption [13]. To exploit such platforms effective power management strategies are needed. The optimization problem is identical to that described in the last paragraph, except that we have a small number of processing units here.

In Section 2 we formally define an optimization problem *Dynamic Power Management (DPM)* that captures the above scenarios. In short, there are m heterogeneous servers (processors). Each server has several states with associated power consumption rates. State transitions incur energy. The planning horizon contains times $t_1 < t_2 < \ldots < t_n$ at which the demand changes. During interval $[t_k, t_{k+1})$ at least d_k servers must be active and available for utilization, $1 \le k \le n-1$. The goal is to find a state transition schedule for the servers minimizing the total energy consumption.

Previous Work. Irani et al. [4] and Augustine et al. [14] study power-down strategies for a single device that is equipped with an active state and several low-power states. The goal is to minimize the energy consumed in an idle period. Our problem DPM is a generalization with multiple, parallel devices and time-dependent demand. The two articles [4, 14] develop online algorithms that achieve optimal competitive ratios. Dynamic power management for a single device with two states is equivalent to the ski-rental problem, a famous rent-or-buy problem [15, 16, 19, 21]. No generalization with several required resources has been examined. Azar et al. [5] study a capital investment problem where machines for manufacturing a product may be purchased over time. The machines differ in the capital and production costs.

Khuller et al. [17] and Li and Khuller [18] introduce machine activation problems that are also motivated by energy conservation in data centers. In [17] the authors assume that there is an activation cost budget, and jobs have to be scheduled on the selected, activated machines so as to minimize the makespan. They present algorithms that simultaneously approximate the budget and the makespan. The second paper [18] considers a generalization where the activation cost of a machine is a non-decreasing function of the load.

In the more applied computer science literature power management strategies and the value of sleep states have been studied extensively. The papers mostly focus on experimental evaluations. Articles that also present analytic results include [9–11, 22]. Ghandi et al. [10] model a server farm with setup costs as an $M/M/m$ queuing system. Lin et al. [22] study a dynamic rightsizing of data centers with homogeneous servers having one sleep state. The operating cost of a server is a convex function of the workload.

Our Contribution. We present an algorithmic study of an important capacity management problem in data centers. Our problem DPM dynamically rightsizes the pool of servers with the objective to minimize the energy consumed. Compared to previous work the new, essential aspects are that we consider (a) a time horizon with varying demand for computing capacity and (b) power-heterogeneous servers. In fact, with homogeneous servers the problem is easy to solve. In DPM the demand for computing capacity is specified by the number of servers needed at any time. In data centers it is common practice that a number of required servers is determined as a function of the current total workload, ignoring specific jobs. DPM focuses on energy conservation instead of individual job placement.

We investigate DPM as an offline problem, i.e. the varying computing demands are known in advance. From an algorithmic point of view it is important to explore the tractability and approximability of the problem. The offline setting is also relevant in practice. Data centers usually analyze past workload traces to identify long-term patterns. The findings are used to specify demands in future time windows.

In Section 3 we study DPM in the scenario that each server has two states, an active state and one sleep state. This is a basic setting that, in a first step, abstracts away the full spectrum of low-power modes. Most of the more applied literature on power management strategies assumes the existence of a single sleep state. We show that DPM can be solved in polynomial time by a combinatorial algorithm. We devise an algorithm that resorts to a single-commodity minimum-cost flow computation. In the corresponding network there is a component for each server. Such a component contains an upper path and a lower path, representing the server's active state and sleep state, respectively. Unfortunately, an arbitrary minimum-cost flow does not correspond to a feasible schedule. Our algorithm modifies flow so that an optimal schedule can be derived.

In Section 4 we investigate DPM in the general scenario that each server has multiple sleep states. We extend our approach based on flow computations. We develop a second algorithm that works with a more complex network in which each component has several lower paths, representing the various low-power states of a server. Furthermore, we need a second commodity to ensure that computing demands are met. With only a single commodity, flow units could switch between lower paths at no cost, and infeasible schedules would result. Given a fractional two-commodity minimum-cost flow, our algorithm executes advanced flow rounding and packing procedures. First, by repeatedly traversing components, the algorithm modifies flow so it becomes integral on the upper paths. Then flow on the lower paths is packed. The final integral flow allows the constructing of a schedule for DPM. Our algorithm achieves an approximation factor of τ, where τ is the number of server types in the problem instance. The servers can be partitioned into τ classes such that, within each class, the servers are identical. Of course, the servers of a class are independent and not synchronized. In practice, a data center has a large collection of machines but a relatively small number of different server architectures. Furthermore, in the optimization, machines with comparable energy consumption characteristics can be assigned to the same server class.

We note that our algorithms can handle the problem extension that the power consumption rates are time-dependent. This can model e.g. scenarios in which servers are temporarily unavailable due to maintenance or because they are reserved for other tasks.

Due to space constraints the proofs of lemmas and propositions are presented in the full version of this paper.

2 PRELIMINARIES

2.1 Problem Definition

We define the optimization problem *Dynamic Power Management (DPM).* A problem instance $\mathcal{I} = (\mathcal{S}, \mathcal{D})$ is specified by a set of servers and varying computing demands over a time horizon. Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a set of *heterogeneous servers.* Each server S_i, $1 \le i \le m$, has an active state as well as one or several standby/sleep states. The states of S_i are denoted by $s_{i,0}, \ldots, s_{i,\sigma_i}$. Here $s_{i,0}$ is the active state and $s_{i,1}, \ldots, s_{i,\sigma_i}$ are the low-power states. The modes have individual power consumption rates. Let $r_{i,j}$ be the power consumption rate of $s_{i,j}$, i.e. $r_{i,j}$ energy units are consumed per time unit while S_i resides in $s_{i,j}$. The states are numbered in order of decreasing rates such that $r_{i,0} > \ldots > r_{i,\sigma_i} \ge 0$. A server can transition between its states. Let $\Delta_{i,j,j'}$ be the non-negative energy needed to move S_i from state $s_{i,j}$ to state $s_{i,j'}$, for any pair $1 \le j, j' \le \sigma_i$. The transition energies satisfy the triangle inequality, i.e. the energy to move directly from $s_{i,j}$ to $s_{i,j'}$ is upper bounded by that of visiting an intermediate state $s_{i,k}$. Formally, $\Delta_{i,j,j'} \le \Delta_{i,j,k} + \Delta_{i,k,j'}$, for any j, j', k.

Over a time horizon the computing demands are given by a *demand profile* $\mathcal{D} = (T, D)$. Tuple $T = (t_1, \ldots, t_n)$ contains the points in time when the computing demands change. There holds $t_1 < t_2 < \ldots < t_n$ so that the time horizon is $[t_1, t_n)$. Tuple $D = (d_1, \ldots, d_{n-1})$ specifies the demands. More precisely, $d_k \in \mathbb{N}_0$ servers are required for computing during interval $[t_k, t_{k+1})$, for any $1 \le k \le n-1$. Thus at least d_k servers must reside in the active state during $[t_k, t_{k+1})$. We have $d_k \leq m$, for any $1 \leq k \leq n-1$, so that the requirements can be met.

Given $\mathcal{I} = (\mathcal{S}, \mathcal{D})$, a *schedule* Σ specifies, for each S_i and any $t \in [t_1, t_n)$, in which state server S_i resides at time t. Schedule Σ is *feasible* if during any interval $[t_k, t_{k+1})$ at least d_k servers are in the active state, $1 \leq k \leq n-1$. The energy $E(\Sigma)$ incurred by Σ is the total energy consumed by all the m servers. Whenever server S_i, $1 \leq i \leq m$, resides in state $s_{i,j}$ it consumes energy at a rate of $r_{i,j}$. Whenever the server transitions from state $s_{i,j}$ to state $s_{i,j'}$, the incurred energy is $\Delta_{i,j,j'}$. The goal is to find an *optimal schedule*, i.e. a feasible schedule Σ that minimizes $E(\Sigma)$. We assume that initially, immediately before t_1, and at time t_n all servers reside in the deepest sleep state, i.e. S_i is in s_{i,σ_i}, $1 \leq i \leq m$. Our algorithms and results can be adapted easily if each server initially/finally takes arbitrary desired states.

2.2 Properties of Optimal Schedules

Given a problem instance $\mathcal{I}$, we characterize optimal schedules. Proposition 2.1 implies that there exists an optimal schedule in which a server never changes state while being in low-power mode. Of course the low-power states may vary for the various intervals in which a server is not active. Proposition 2.2 states that there exists an optimal schedule executing state transitions only when the computing demands change. A server *powers up* if it transitions from a low-power state to the active state. A server *powers down* if it moves from the active state to a low-power state.

Proposition 2.1. *There exists an optimal schedule with the following property. Suppose that S_i powers down at time t and next powers up at time t'. Then between t and t' S_i resides in a single state $s_{i,j}$, where $j > 0$. At time t S_i transitions directly from $s_{i,0}$ to $s_{i,j}$. At time t' it moves directly from $s_{i,j}$ to $s_{i,0}$.*

Proposition 2.2. *There exists an optimal schedule that satisfies the property of Proposition 2.1 and performs state transitions only at the times of T.*

We finally argue that w.l.o.g. the power-down energies $\Delta_{i,0,j}$ are equal to 0, $1 \leq i \leq m$ and $1 \leq j \leq \sigma_i$. We will always focus on optimal schedules with the property given in Proposition 2.1. At times t_1 and t_n every server is in its deepest sleep state. The first time server S_i moves to the active state, the least energy is consumed if it transitions directly from s_{i,σ_i} to $s_{i,0}$. The last time S_i powers down, the best option is to move directly from $s_{i,0}$ to s_{i,σ_i}. Hence, every server S_i performs the same number of transitions from $s_{i,0}$ to $s_{i,j}$ as from $s_{i,j}$ to $s_{i,0}$, for any $1 \leq j \leq \sigma_i$. For any server S_i, only energies $\Delta_{i,0,j}$ and $\Delta_{i,j,0}$, $1 \leq j \leq \sigma_i$, are relevant. Therefore, if $\Delta_{i,0,j} > 0$, we can add this energy to $\Delta_{i,j,0}$, i.e. $\Delta'_{i,j,0} := \Delta_{i,0,j} + \Delta_{i,j,0}$ and $\Delta'_{i,0,j} := 0$.

3 SERVERS WITH TWO STATES

We study the variant of DPM in which each server S_i has exactly two states, an active state $s_{i,0}$ and a sleep state $s_{i,1}$, $1 \leq i \leq m$.

Theorem 3.1. *Let $\mathcal{I}$ be an instance of DPM in which each server has exactly two states. An optimal schedule for $\mathcal{I}$ can be computed in polynomial time by a combinatorial algorithm that uses a minimum-cost flow computation.*

In the remainder of this section we prove Theorem 3.1. We first show that we may assume w.l.o.g. that the power consumption rates in the sleep states are equal to 0. More specifically, for any problem instance $\mathcal{I}$, an optimal schedule can be derived from an optimal solution to a modified instance $\mathcal{I}'$ in which the power consumption rates in the sleep states are indeed 0. Formally, given $\mathcal{I} = (\mathcal{S}, \mathcal{D})$, define an instance $\mathcal{I}' = (\mathcal{S}', \mathcal{D})$. Set $\mathcal{S}'$ consists of servers $S'_1, \ldots, S'_m$, where each server S'_i has again an active state and a sleep state. For any S'_i, let $r'_{i,0} = r_{i,0} - r_{i,1}$ and $r'_{i,1} = 0$, i.e. the rates are reduced by $r_{i,1}$. All other problem parameters of $\mathcal{I}'$, namely the state transition energies and the demand profile, are identical to those of $\mathcal{I}$. The next proposition states that an optimal schedule for $\mathcal{I}$ translates to an optimal schedule for $\mathcal{I}'$ and vice versa. Only the consumed energy differs by $\sum_{i=1}^{m} r_{i,1}(t_n - t_1)$.

Proposition 3.2. *Any schedule Σ for $\mathcal{I}$ that is executed for $\mathcal{I}'$ consumes an energy of $E(\Sigma) - \sum_{i=1}^{m} r_{i,1}(t_n - t_1)$. Any schedule Σ' for $\mathcal{I}'$ that is executed for $\mathcal{I}$ consumes an energy of $E(\Sigma') + \sum_{i=1}^{m} r_{i,1}(t_n - t_1)$.*

In the following let $\mathcal{I} = (\mathcal{S}, \mathcal{D})$ be a problem instance in which the power consumption rates in the servers' sleep states are 0. To simplify notation let $r_i := r_{i,0}$ be the power consumption rate of S_i in the active state, $1 \leq i \leq m$. Moreover, let $\Delta_i := \Delta_{i,1,0}$ be the energy needed to transition S_i from the sleep state to the active state. We develop an algorithm $\mathcal{A}_1$ that computes an optimal schedule. Based on Proposition 2.2, we focus on schedules that perform state transitions only at the times of T. Given $\mathcal{I} = (\mathcal{S}, \mathcal{D})$, $\mathcal{A}_1$ constructs a network $\mathcal{N}(\mathcal{I})$. Any feasible schedule Σ for $\mathcal{I}$ translates to a feasible flow of cost $E(\Sigma)$ in $\mathcal{N}(\mathcal{I})$. Any feasible flow of cost C in $\mathcal{N}(\mathcal{I})$ can be converted so that it corresponds to a feasible schedule consuming energy C. The conversion requires some work but can be performed in a polynomial number of steps.

3.1 Construction of the Network

Consider any problem instance $\mathcal{I} = (\mathcal{S}, \mathcal{D})$.

Network components. Network $\mathcal{N}(\mathcal{I})$ contains a *component* C_i, for each server S_i, $1 \leq i \leq m$. Such a component C_i, which is depicted in Figure 1, consists of an *upper path* and a *lower path*. The upper path represents the active state of S_i; the lower path models the server's sleep state. The computing demands change at the times $t_1 < \ldots < t_n$ in T. For any t_k, $1 \leq k \leq n$, there is a vertex $u_{i,k}$ on the upper path. Vertices $u_{i,k}$ and $u_{i,k+1}$ are connected by a directed edge $(u_{i,k}, u_{i,k+1})$ of cost $r_i(t_{k+1} - t_k)$, $1 \leq k \leq n-1$. This cost is equal to the energy consumed if S_i is in the active state during $[t_k, t_{k+1})$. Similarly, for any t_k, $1 \leq k \leq n$, there is a vertex $l_{i,k}$ on the lower path. In order to ensure that at least d_k servers are in the active state during $[t_k, t_{k+1})$, if $k < n$, we need two auxiliary vertices $l^a_{i,k}$ and $l^b_{i,k}$. These vertices are again connected by directed edges. There is an edge $(l_{i,k}, l^a_{i,k})$, followed by two edges $(l^a_{i,k}, l^b_{i,k})$ and $(l^b_{i,k}, l_{i,k+1})$, for any k with $1 \leq k \leq n-1$. The cost of each of these edges is 0 because the energy consumption in the sleep state is 0.

The lower and the upper path are connected by additional edges that model state transitions. Recall that all servers are in the sleep state at times t_1 and t_n. For any k with $1 \leq k \leq n-1$, there is a

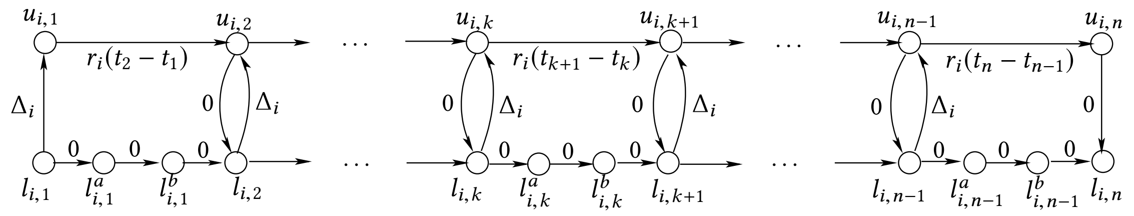

Figure 1: The component C_i for server S_i

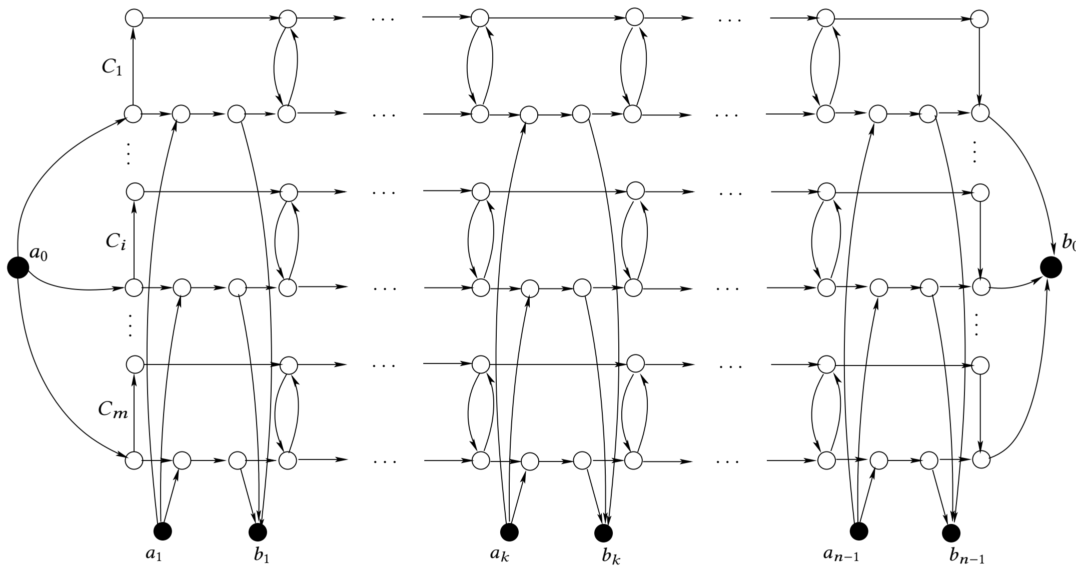

Figure 2: The network $\mathcal{N}(\mathcal{I})$

directed edge $(l_{i,k}, u_{i,k})$ of cost Δ_i, representing a power-up operation of S_i at time t_k. For any k with $1 < k \leq n$, there is a directed edge $(u_{i,k}, l_{i,k})$ of cost 0, modeling a power-down operation of S_i at time t_k. The capacity of each edge of C_i is equal to 1.

The entire network. In $\mathcal{N}(\mathcal{I})$ components $C_1, \ldots, C_m$ are aligned in parallel and connected to a source a_0 and a sink b_0. The general structure of $\mathcal{N}(\mathcal{I})$ is depicted in Figure 2. There is a directed edge from a_0 to $l_{i,1}$ in C_i, for any $1 \leq i \leq m$. Furthermore, there is a directed edge from $l_{i,n}$ to b_0, for any $1 \leq i \leq m$. Each of these edges has a cost of 0 and a capacity of 1. Vertex a_0 has a supply of m, and b_0 has a demand of m. Hence m units of flow must be shipped through $C_1, \ldots, C_m$. Since all edges have a capacity of 1, one unit of flow must be routed through each C_i, $1 \leq i \leq m$. Whenever the unit traverses the upper path, S_i is in the active state. Whenever the unit traverses the lower path, S_i is in the sleep state.

In order to ensure that at least d_k servers are in the active state during $[t_k, t_{k+1})$, $1 \leq k \leq n-1$, we introduce additional sources and sinks. Network $\mathcal{N}(\mathcal{I})$ has a source a_k and a sink b_k with supply/demand d_k, for any $1 \leq k \leq n-1$. There is a directed edge from a_k to $l^a_{i,k}$ on the lower path of each C_i, $1 \leq i \leq m$. Furthermore, there is a directed edge from each $l^b_{i,k}$ to b_k, $1 \leq i \leq m$. The cost and capacity of each of these edges is equal to 0 and 1, respectively. Since d_k flow units have to be shipped from a_k to b_k, there must exist at least d_k components C_i in which the flow unit from a_0 to b_0 traverses the upper path from $u_{i,k}$ to $u_{i,k+1}$. Hence the corresponding servers are in the active state during $[t_k, t_{k+1})$. The encoding length of $\mathcal{N}(\mathcal{I})$ is polynomial in that of $\mathcal{I}$.

LEMMA 3.3. *Any feasible schedule Σ in which state transitions are performed only at the times of T corresponds to a feasible flow of cost $E(\Sigma)$ in $\mathcal{N}(\mathcal{I})$.*

3.2 Analysis of Flows

We analyze feasible flows in $\mathcal{N}(\mathcal{I})$. The goal is to show that any feasible flow f can be converted into one that corresponds to a feasible schedule Σ for $\mathcal{I}$; the energy consumed by Σ will be equal to the cost of f. The conversion is not immediate. A feasible flow might not be well-behaved, i.e. flow shipped out of a source a_k is not necessarily routed to b_k, $0 \leq k \leq n-1$. It may happen that flow leaving a_k is routed to a sink $b_{k'}$, where $k' > k$, or to b_0.

In $\mathcal{N}(\mathcal{I})$ all edge capacities and supplies/demands are integer values. Hence in $\mathcal{N}(\mathcal{I})$ there exists a minimum-cost flow that is integral. A flow f is called integral if the flow $f(e)$ along any edge e takes an integer value. Moreover, there exist polynomial time combinatorial algorithms that compute an integral minimum-cost flow, given a network with integer edge capacities and supplies/demands, see [2].

We will always work with a flow f in $\mathcal{N}(\mathcal{I})$ that is integral. Such a flow translates into a state transition schedule for the servers if,

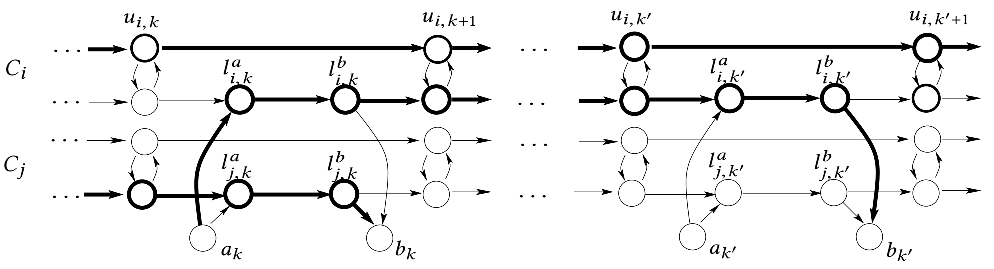

Figure 3: A flow that is not consistent in $[t_{k+1}, t_{k+2})$.

for each C_i and each k, one flow unit traverses either the upper path from $u_{i,k}$ to $u_{i,k+1}$ or the lower path from $l_{i,k}$ to $l_{i,k+1}$. Formally we call an integral flow *consistent in* $[t_k, t_{k+1})$, where $1 \le k \le n-1$, if $f(u_{i,k}, u_{i,k+1}) + f(l_{i,k}, l^a_{i,k}) = 1$ holds for all $i = 1, \ldots, m$. In this definition we only consider flow from $l_{i,k}$ to $l^a_{i,k}$. This will be sufficient for our purposes. An integral flow is called *consistent* if it is consistent in all intervals $[t_k, t_{k+1})$, $1 \le k \le n-1$. In the following we will prove that any feasible integral flow can be converted into one that is consistent. The next lemma identifies properties of feasible flow. Part b) characterizes the shipment of flow that is not consistent and will allow us to generate flow that satisfies consistency.

Lemma 3.4. *Let f be any feasible integral flow in $\mathcal{N}(\mathcal{I})$.*

a) f is consistent in $[t_1, t_2)$.
b) Suppose that f is consistent in $[t_1, t_2), \ldots, [t_k, t_{k+1})$ but not in $[t_{k+1}, t_{k+2})$. Then there exist components C_i and C_j such that $f(u_{i,k+1}, u_{i,k+2}) + f(l_{i,k+1}, l^a_{i,k+1}) = 2$ and $f(u_{j,k+1}, u_{j,k+2}) + f(l_{j,k+1}, l^a_{j,k+1}) = 0$, cf. Figure 3. In C_i there holds $f(u_{i,k}, u_{i,k+1}) = 1$. One flow unit is shipped from source a_k to $l^a_{i,k}$ and is further routed to $l_{i,k+1}$. In C_j there holds $f(l_{j,k}, l^a_{j,k}) = 1$. This unit is routed via $l^b_{j,k}$ to sink b_k.

3.3 Making a Flow Consistent

Let f be a feasible integral flow in $\mathcal{N}(\mathcal{I})$. We describe how algorithm $\mathcal{A}_1$ modifies f so that the resulting flow is consistent. By Lemma 3.4a), f is consistent in $[t_1, t_2)$. Suppose that f is consistent in $[t_1, t_2), \ldots, [t_k, t_{k+1})$ but not in $[t_{k+1}, t_{k+2})$. $\mathcal{A}_1$ modifies the flow so that it fulfills consistency in $[t_1, t_2), \ldots, [t_{k+1}, t_{k+2})$. The modifications are performed sequentially for all further intervals.

Modifying flow: By assumption, f is consistent in $[t_1, t_2), \ldots, [t_k, t_{k+1})$ but not in $[t_{k+1}, t_{k+2})$. Hence there must exist components C_i and C_j with the properties specified in Lemma 3.4b), see again Figure 3. In C_i a total of two flow units leave $u_{i,k+1}$ and $l_{i,k+1}$ along the upper and lower paths, respectively. On the upper path one flow unit traverses the edge from $u_{i,k}$ and $u_{i,k+1}$. On the lower path one unit is injected from a_k. This unit reaches $l^a_{i,k}$ and continues via $l^b_{i,k}$ to $l_{i,k+1}$. In C_j no flow leaves $u_{j,k+1}$ or $l_{j,k+1}$. A flow unit is shipped from $l_{j,k}$ to $l^a_{j,k}$ and this unit is routed to sink b_k via $l^b_{j,k}$.

While there exist components C_i and C_j as specified above, $\mathcal{A}_1$ works as follows. It determines the smallest integer k', with $k' > k$, such that a flow unit is routed from C_i to sink $b_{k'}$, i.e. $f(l^b_{i,k'}, b_{k'}) = 1$. Such an integer must exist since otherwise a total of two flow units must reach the end of C_i at $u_{i,n}$ and $l_{i,n}$. These two flow units cannot feasibly be routed to b_0 along the unit-capacity edge $(l_{i,n}, b_0)$. Let $P_i(k, k')$ be the path from $l^b_{i,k}$ to $l^b_{i,k'}$ that uses only edges of the lower path of C_i. All edges of $P_i(k, k')$ carry one unit of flow. Similarly, let $P_j(k, k')$ be the path from $l^b_{j,k}$ to $l^b_{j,k'}$ that uses only edges of the lower path of C_j. In the flow modification there are two cases depending on whether or not $P_j(k, k')$ carries flow.

Flow modification, type 1: Suppose that $P_j(k, k')$ does not ship any flow, see Figure 3. Loosely speaking, $\mathcal{A}_1$ replaces flow along $P_i(k, k')$ by flow on $P_j(k, k')$. Formally, the modified flow is as follows. In C_i the flow unit entering $l^b_{i,k}$ is routed to b_k, i.e. $f'(l^b_{i,k}, b_k) = 1$. In C_j algorithm $\mathcal{A}_1$ removes the flow unit leaving $l^b_{j,k}$, i.e. $f'(l^b_{j,k}, b_k) = 0$. For all edges e of $P_i(k, k')$, the algorithm sets $f'(e) = 0$. For all edges e of $P_j(k, k')$, it sets $f'(e) = 1$. Finally it removes the flow unit leaving $l^b_{i,k'}$, i.e. $f'(l^b_{i,k'}, b_{k'}) = 0$, and routes one unit from $l^b_{j,k'}$ to $b_{k'}$, i.e. $f'(l^b_{j,k'}, b_{k'}) = 1$. For all the other edges not considered here, the flow remains unchanged. Obviously, after these modifications, the amount of flow routed into b_k and $b_{k'}$ has not change. The flow conservation law is observed at all vertices of $P_i(k, k')$ and $P_j(k, k')$. Hence the new flow is feasible. Furthermore, the cost of the flow has not changed because the flow update only affects edges of cost 0. Note that $f'(l_{i,k+1}, l^a_{i,k+1}) = 0$ and $f'(l_{j,k+1}, l^a_{j,k+1}) = 1$. Hence restricted to C_i and C_j the new flow is consistent in $[t_{k+1}, t_{k+2})$.

Flow modification, type 2: Assume that some edge of $P_j(k, k')$ carries flow. Then this flow must enter C_j from some source among $a_{k+1}, \ldots, a_{k'}$. $\mathcal{A}_1$ determines the smallest integer k^*, with $k^* > k$, such that $f(a_{k^*}, l^a_{j,k^*}) = 1$. There holds $k^* \le k'$. In component C_i the corresponding edge (a_{k^*}, l^a_{i,k^*}) does not ship flow because all edges of $P_i(k, k')$ carry one unit of flow and no further unit can be injected from a_{k^*}. Let $P_i(k, k^*)$ be the path from $l^b_{i,k}$ to l^a_{i,k^*} that uses only edges of the lower path of C_i. Analogously, let $P_j(k, k^*)$ be the path from $l^b_{j,k}$ to l^a_{j,k^*} that uses only edges of the lower path of C_j. $\mathcal{A}_1$ replaces flow on $P_i(k, k^*)$ by flow on $P_j(k, k^*)$. More specifically, the flow unit routed into $l_{i,k}$ is shipped to b_k, i.e. $f'(l^b_{i,k}, b_k) = 1$. The flow unit on edge $(l^b_{j,k}, b_k)$ is removed. For all edges e of $P_i(k, k^*)$, $\mathcal{A}_1$ sets $f'(e) = 0$. For all edges e of $P_j(k, k^*)$, it sets $f'(e) = 1$. Finally, it sets $f'(a_{k^*}, l^a_{i,k^*}) = 1$ and $f'(a_{k^*}, l^a_{j,k^*}) = 0$. The new flow is feasible, and during the modification the cost has not changed. Restricted to C_i and C_j the new flow is consistent in $[t_{k+1}, t_{k+2})$ because $f'(l_{i,k+1}, l^a_{i,k+1}) = 0$ and $f'(l_{j,k+1}, l^a_{j,k+1}) = 1$.

The above flow modifications do not change flow in components other than C_i and C_j. By repeating the flow update operations for other pairs of network components violating consistency, $\mathcal{A}_1$ obtains a flow that is consistent in $[t_1, t_2), \ldots, [t_{k+1}, t_{k+2})$. The total number of steps to perform the modifications is polynomial in $\mathcal{N}(\mathcal{I})$. The next lemma summarizes the result.

Lemma 3.5. *Let f be a feasible integral flow of cost C in $\mathcal{N}(\mathcal{I})$. Then f can be transformed into a feasible integral flow that is consistent and has cost C. The transformation takes polynomial time.*

3.4 Establishing the Theorem

The next lemma states that a feasible consistent flow properly ships flow from sources to sinks.

Lemma 3.6. *In any feasible integral flow f that is consistent, all flow leaving a_k is routed to b_k, $1 \leq k \leq n-1$.*

We finish the proof of Theorem 3.1. Given problem instance $\mathcal{I}$, $\mathcal{A}_1$ constructs $\mathcal{N}(\mathcal{I})$ and computes an integral minimum-cost flow f^* using a combinatorial algorithm. Executing the flow modifications described above, the algorithm obtains an integral minimum-cost flow f that is consistent. Lemma 3.6 implies that in f all flow units leaving a_0 are transferred to b_0. By the edge capacity constraints, one unit of flow is transferred through each C_i, $1 \leq i \leq m$.

$\mathcal{A}_1$ derives a schedule Σ for $\mathcal{I}$ by keeping track of these flow units in $C_1, \ldots, C_m$. Consider component C_i, $1 \leq i \leq m$. While the flow unit traverses the upper path, server S_i is in the active state. While the flow unit traverses the lower path, S_i is in the sleep state. If the flow traverses an edge $(l_{i,k}, u_{i,k})$, S_i powers up at time t_k. If the flow traverses $(u_{i,k}, l_{i,k})$, the server powers down at time t_k. The energy consumed by S_i is exactly equal to the cost incurred by the flow unit traversing C_i. Hence the energy consumed by Σ is equal to the cost of f, and this is equal to the cost of f^*.

It remains to verify that Σ is feasible. By Lemma 3.6, in f all flow units leaving a_k are shipped to b_k, $1 \leq k \leq n-1$. Consider any fixed k, $1 \leq k \leq n-1$. There must exist d_k components C_i such that a flow unit is routed from a_k to $l^a_{i,k}$ and further on to $l^b_{i,k}$ and b_k. Since f is consistent in $[t_k, t_{k+1})$, there holds $f(u_{i,k}, u_{i,k+1}) = 1$. If $f(l_{i,k}, l^a_{i,k}) = 1$, then two units of flow would leave $l^a_{i,k}$, violating the capacity of the outgoing edge. Thus in $[t_k, t_{k+1})$ at least d_k servers are in the active state. Optimality of Σ follows from Proposition 2.2 and Lemma 3.3.

4 SERVERS WITH MULTIPLE STATES

We develop an approximation algorithm for DPM in the general setting that each server may have an arbitrary number of states. Let $\mathcal{I} = (\mathcal{S}, \mathcal{D})$ be an input with τ server types, i.e. each server of $\mathcal{S}$ belongs to one of τ classes, where $\tau \in \mathbb{N}$. Formally, $\mathcal{S}$ is partitioned into $\mathcal{S}_1, \ldots, \mathcal{S}_\tau$. Within each server type/class $\mathcal{S}_i$, $1 \leq i \leq \tau$, all servers are identical. Every server of $\mathcal{S}_i$ has $\sigma_i + 1$ states $s_{i,0}, \ldots, s_{i,\sigma_i}$ with power consumption rates $r_{i,0} > \ldots > r_{i,\sigma_i}$. Here $s_{i,0}$ is again the active state; the other states are low-power modes. The energy needed to transition from $s_{i,j}$ to $s_{i,0}$ is denoted by $\Delta_{i,j}$, $1 \leq j \leq \sigma_i$. The state transition energy from the active state to any lower-power state is 0. The servers of $\mathcal{S}_i$ are independent and not synchronized. Over the time horizon each server may reside in individual states and perform state transitions independent of the other servers. Let m_i be the number of servers in $\mathcal{S}_i$. There holds $\sum_{i=1}^{\tau} m_i = m$.

Theorem 4.1. *Let $\mathcal{I}$ be an instance of DPM with τ server types. A schedule whose energy consumption is at most τ times the minimum one for $\mathcal{I}$ can be computed in polynomial time based on a min-cost two-commodity flow computation.*

In the remainder of this section we develop an algorithm $\mathcal{A}_2$ that, given $\mathcal{I} = (\mathcal{S}, \mathcal{D})$, constructs a feasible schedule attaining a τ-approximation on the consumed energy. This establishes Theorem 4.1. By Proposition 2.2 we restrict ourselves to schedules with the following two properties. While a server is in low-power mode, it uses a single state. State transitions are performed only at the times of T.

Algorithm $\mathcal{A}_2$ constructs a network $\mathcal{N}(\mathcal{I})$. Compared to the construction in Section 3, the main differences are as follows. Each network component will represent a class of servers so that the encoding length of $\mathcal{N}(\mathcal{I})$ is polynomial in that of $\mathcal{I}$. A component has a collection of lower paths corresponding to the various low-power modes of the servers. We need a second commodity to ensure that computing demands are met. This will allow us to reduce the number of auxiliary vertices on the lower paths.

Given $\mathcal{N}(\mathcal{I})$, $\mathcal{A}_2$ computes a minimum-cost flow f^*. Since the network has two commodities, f^* is not integral but fractional in general. In a sequence of rounding and packing operations $\mathcal{A}_2$ transforms f^* into an integral one that guides the construction a feasible schedule for $\mathcal{I}$. The cost of the integral flow and the constructed schedule will be at most τ times that of f^*.

4.1 Construction of the Network

We describe $\mathcal{N}(\mathcal{I})$, given $\mathcal{I} = (\mathcal{S}, \mathcal{D})$.

Network components with multiple paths. For every server type i, the network contains a component C_i, $1 \leq i \leq \tau$. The component represents all the m_i servers of $\mathcal{S}_i$. Exactly m_i flow units will be routed through C_i, modeling the individual states and actions of the servers. Component C_i consists of an *upper path* and σ_i *lower paths*. The general structure is depicted in Figure 4. We search for an optimal schedule in which state transitions are performed only at times $t_1 < \ldots < t_n$ in T, cf. Proposition 2.2. In C_i the upper path corresponds to the active state of the servers of $\mathcal{S}_i$. For any t_k, $1 \leq k \leq n$, there is a vertex $u_{i,k}$ on the upper path. Vertices $u_{i,k}$ and $u_{i,k+1}$ are connected by a directed edge $(u_{i,k}, u_{i,k+1})$ of cost $r_{i,0}(t_{k+1} - t_k)$, $1 \leq k \leq n-1$. The cost is equal to energy consumed by one server of $\mathcal{S}_i$ if it resides in the active state during $[t_k, t_{k+1})$. The capacity of $(u_{i,k}, u_{i,k+1})$ and in fact of all edges of C_i is equal to m_i, reflecting that C_i represents m_i servers in $\mathcal{S}_i$.

The σ_i lower paths correspond to the σ_i low-power states. Consider any j with $1 \leq j \leq \sigma_i$. On lower path j there is a vertex $l_{i,j,k}$ and an auxiliary vertex $l^a_{i,j,k}$, for any $1 \leq k \leq n-1$. Moreover, there is a final vertex $l_{i,j,n}$. The auxiliary vertices will help to ensure that a total of at least d_k flow units traverse the edges $(u_{i,k}, u_{i,k+1})$ on the upper paths, considering all the components C_i, $1 \leq i \leq \tau$. We

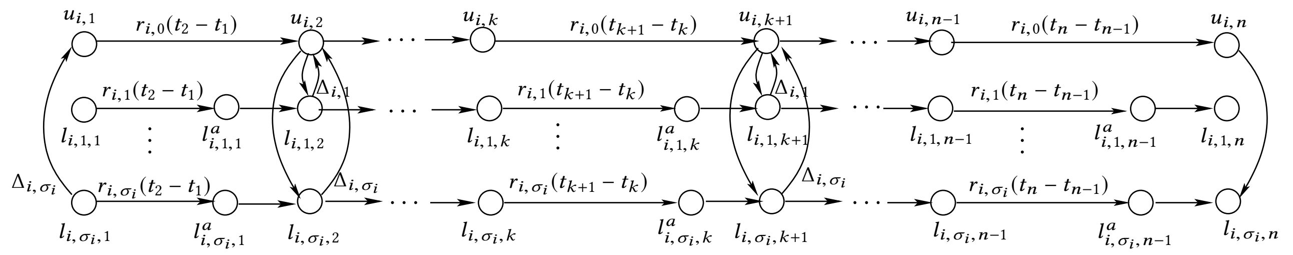

Figure 4: The component C_i for server type i. Unlabeled edges connecting two vertices have cost 0.

do not need a second auxiliary vertex $l^b_{i,j,k}$ because we work with two commodities. On lower path j the vertices are connected as follows. For any k, where $1 \le k \le n-1$, there is a directed edge $(l_{i,j,k}, l^a_{i,j,k})$ of cost $r_{i,j}(t_{k+1} - t_k)$, representing the energy consumed by a server if it is in state $s_{i,j}$ during $[t_k, t_{k+1})$. Furthermore there is an edge $(l^a_{i,j,k}, l_{i,j,k+1})$ of cost 0.

The upper path is connected to the lower paths by additional edges that model state transitions. We assume that at times t_1 and t_n all servers of $\mathcal{S}_i$ are in the deepest low-power state s_{i,σ_i}. Thus there is a directed edge $(l_{i,\sigma_i,1}, u_{i,1})$ of cost Δ_{i,σ_i} modeling possible power-up operations of servers at time t_1. Furthermore there is a directed edge $(u_{i,n}, l_{i,\sigma_i,n})$ of cost 0 representing power-down operations at time t_n. For any $1 < k < n$ and any $1 \le j \le \sigma_i$, there is a directed edge $(u_{i,k}, l_{i,j,k})$ of cost 0 and a directed edge $(l_{i,j,k}, u_{i,k})$ of cost $\Delta_{i,j}$. Since we consider schedules specified in Proposition 2.1, there are no state transitions among low-power states; thus there are no edges between the lower paths. (We remark that on lower path j, $1 \le j < \sigma_i$, we could remove the first and the last vertex but it is not important.) Note again that the capacity of each edge of C_i is m_i.

The network with two commodities. In $\mathcal{N}(\mathcal{I})$ components $C_1, \ldots, C_\tau$ are aligned in parallel and connected to vertices a_0 and b_0. The general composition is similar to that depicted in Figure 2; an accurate figure is given in the full paper. Vertices a_0 and b_0 inject and absorb flow of commodity 1. Specifically, a_0 has a supply of m and b_0 has a demand of m of commodity 1. The connections are as follows. At times t_1 and t_n the servers are in the deepest low-power mode. Hence, for any $1 \le i \le \tau$, there exist directed edges $(a_0, l_{i,\sigma_i,1})$ and $(l_{i,\sigma_i,n}, b_0)$. Each of these edges has a cost of 0 and a capacity of m_i so that m_i flow units can be routed from a_0 to b_0 via C_i.

Network $\mathcal{N}(\mathcal{I})$ contains further sources and sinks that inject and absorb flow of commodity 2. This second commodity will ensure that the computing demands are met. Consider any k with $1 \le k \le n-1$. There is a source a_k and a sink b_k with a supply/demand of $D_k = \sum_{i=1}^{\tau} m_i(\sigma_i - 1) + d_k$ of commodity 2. Vertices a_k and b_k are connected to all lower paths in the components. For any i and j with $1 \le i \le \tau$ and $1 \le j \le \sigma_i$, there is a directed edge $(a_k, l^a_{i,j,k})$ into the auxiliary vertex on lower path j in C_i. Moreover, there is a directed edge $(l_{i,j,k+1}, b_k)$ from the following vertex on the lower path into b_k. Each of these edges has a cost of 0 and a capacity of m_i. Lemma 4.2 below states that in any feasible flow, for any $1 \le k \le n-1$, at least d_k units will be shipped along the edges $(u_{i,k}, u_{i,k+1})$ on the upper paths of the components $C_1, \ldots, C_\tau$.

So far we have specified the total capacity of any edge in $\mathcal{N}(\mathcal{I})$. It remains to specify edge capacity constraints for the two commodities. Consider any $1 \le i \le \tau$. For any edge of C_i, the capacity of commodity 1 is m_i. The same holds true for the edges $(a_0, l_{i,\sigma_i,1})$ and $(l_{i,\sigma_i,n}, b_0)$. On all the edges leaving a_k or entering b_k, $1 \le k \le n-1$, the capacity of commodity 1 is equal to 0. Hence flow of commodity 1 must not traverse these edges. In the network components commodity 2 may only traverse the edges from the auxiliary vertices to the subsequent vertices on the lower paths. Hence commodity 2 has a capacity constraint of m_i on each of the edges $(a_k, l^a_{i,j,k})$, $(l^a_{i,j,k}, l_{i,j,k+1})$ and $(l_{i,j,k+1}, b_k)$, where $1 \le i \le \tau$, $1 \le j \le \sigma_i$ and $1 \le k \le n-1$. For all the other edges in $\mathcal{N}(\mathcal{I})$, commodity 2 has a capacity of 0. Hence all flow from a_k, $1 \le k \le n-1$, must be shipped to b_k via edges $(l^a_{i,j,k}, l_{i,j,k+1})$. The encoding length of $\mathcal{N}(\mathcal{I})$ is polynomial in that of $\mathcal{I}$ because the m_i identical servers of $\mathcal{S}_i$ are modeled by a single component C_i, $1 \le i \le \tau$.

Lemma 4.2, as mentioned above, identifies an important property of feasible flow. Lemma 4.3 states that every optimal schedule with the properties of Proposition 2.2 translates to a feasible flow with the same energy/cost.

LEMMA 4.2. *In $\mathcal{N}(\mathcal{I})$ there exists a feasible flow. Any feasible flow f satisfies $\sum_{i=1}^{\tau} f(u_{i,k}, u_{i,k+1}) \ge d_k$, for $1 \le k \le n-1$.*

LEMMA 4.3. *Let Σ be an optimal schedule as specified in Proposition 2.2. Then Σ corresponds to a feasible flow of cost $E(\Sigma)$ in $\mathcal{N}(\mathcal{I})$.*

4.2 Algorithm Outline & Flow Properties

Given $\mathcal{N}(\mathcal{I})$, algorithm $\mathcal{A}_2$ computes a feasible minimum-cost flow f^*. By Lemma 4.3 the cost of f^*, denoted by $cost(f^*)$, is a lower bound on the energy consumed by an optimal schedule for $\mathcal{I}$. Since f^* involves two commodities, it is fractional in general. In particular, it may be fractional on the upper paths of the components. On the corresponding edges the flow has to be raised, for sufficiently many components, so that a feasible schedule for $\mathcal{I}$ can be derived later. $\mathcal{A}_2$ modifies f^* in three main steps. The resulting flow will be integral. (1) First $\mathcal{A}_2$ scales f^* by a factor of τ. (2) Then $\mathcal{A}_2$ modifies the scaled flow so that it becomes integral on the upper paths of the components. Specifically, on edge $(u_{i,k}, u_{i,k+1})$ exactly $d_{i,k} = \min\{m_i, \lfloor \tau f^*(u_{i,k}, u_{i,k+1}) \rfloor\}$ units of flow are routed, where $1 \le i \le \tau$ and $1 \le k \le n-1$. Lemma 4.4 below states that $\sum_{i=1}^{\tau} d_{i,k} \ge d_k$, for any $1 \le k \le n-1$. This property will later admit the construction of a feasible schedule in which the computing demands of $\mathcal{I}$ are met. (3) Given the flow of Step 2, $\mathcal{A}_2$ packs fractional flows on the lower paths of the components $C_1, \ldots, C_\tau$. Using the integral flow obtained in Step 3, $\mathcal{A}_2$ constructs a feasible schedule for $\mathcal{I}$ whose energy consumption is upper bounded by the cost of that flow. Once f^* has been scaled in Step 1, the subsequent flow modifications of Steps 2 and 3 never increase cost. Thus the energy consumed by the schedule is at most $\tau\, cost(f^*)$.

Lemma 4.4. *For* $k = 1, \ldots, n-1$, $\sum_{i=1}^{\tau} d_{i,k} \geq d_k$.

Given Lemma 4.4, a natural idea for finding an integral solution is to use a flow computation: Determine a single-commodity minimum-cost flow that ships m_i flow units through component C_i and, importantly, exactly $d_{i,k}$ units along edge $(u_{i,k}, u_{i,k+1})$, where $1 \leq i \leq \tau$ and $1 \leq k \leq n-1$. However, one has to prove that the cost of such a flow is upper bounded by $\tau\, cost(f^*)$. Such a proof involves arguments and flow modifications contained in Steps 1 and 2 of $\mathcal{A}_2$. Therefore we describe them explicitly as algorithmic steps. Step 3 could indeed be replaced by a min-cost flow computation. However, we instead devise a faster $O(n^2 \sum_{i=1}^{\tau} \sigma_i)$ time routine for constructing an integral flow along the lower paths of the components.

In the following, when describing flow modifications, we will always focus on one particular network component. All flow updates will be performed independently for the components. Hence in the corresponding exposition, we consider an arbitrary but fixed component $C = C_i$, $1 \leq i \leq \tau$. This allows us to simplify notation and omit the index i. On the upper path the vertices are $u_1, \ldots, u_n$. Component C has $\sigma = \sigma_i$ lower paths. On lower path j, $1 \leq j \leq \sigma$, the vertices are $l_{j,k}$ and $l^a_{j,k}$, for $k = 1, \ldots, n-1$, followed by the final vertex $l_{j,n}$. Let $m_c = m_i$ be the number of servers in class $\mathcal{S}_i$ represented by $C = C_i$.

Nested structure of flows. We show that in each network component C flow f^* has a crucial property, i.e. it exhibits a nested structure. Let $P_j(k, k')$ be the path from u_k to $u_{k'}$ along lower path j, where $1 \leq j \leq \sigma$ and $1 < k < k' < n$. More specifically, the path consists of $(u_k, l_{j,k})$, followed by the path from $l_{j,k}$ to $l_{j,k'}$ on lower path j, followed by $(l_{j,k'}, u_{k'})$. For $k = 1$ and $1 < k' < n$, we define $P_\sigma(1, k')$ as the path consisting of the edges from $l_{\sigma,1}$ to $l_{\sigma,k'}$ on lower path σ, followed by the edge $(l_{\sigma,k'}, u_{k'})$. For $k' = n$ and $1 < k < n$, path $P_\sigma(k, n)$ consists of edge $(u_k, l_{\sigma,k})$, followed by the edges from $l_{\sigma,k}$ to $l_{\sigma,n}$ on lower path σ. Finally $P(k, k')$ is the path connecting u_k and $u_{k'}$ on the upper path of the component, for any $1 \leq k < k' \leq n$. In the sequel, unless otherwise stated, flow always refers to commodity 1. Consider any path P. We say that P *routes flow* if, for any edge of P, the flow is strictly positive.

The following property of a flow will be important. A flow f in component C is *nested* if it satisfies the following condition. Let $P_i(k_1, k_2)$ and $P_j(k_3, k_4)$ be two paths such that both route flow and $i < j$. Then one of the relations (a–c) holds: (a) $k_2 < k_3$; (b) $k_4 < k_1$; or (c) $k_3 \leq k_1 < k_2 \leq k_4$. Intuitively, the endpoints of the two paths do not alternate. Both endpoints of $P_i(k_1, k_2)$ occur either before, after or in between those of $P_j(k_3, k_4)$.

Lemma 4.5. *In each component C flow f^* is nested.*

Loop-freeness. Given f^*, $\mathcal{A}_2$ slightly modifies it so that it becomes *loop-free* in each C. A flow is loop-free in C if there exists no vertex u_k such that edges $(l_{i,k}, u_k)$ and $(u_k, l_{j,k})$ both route flow, where $1 \leq i, j \leq \sigma$. Suppose that there exists such a vertex u_k. Since f^* is nested, $i = j$ must hold. Hence $\mathcal{A}_2$ can simply remove $\min\{f^*(l_{i,k}, u_k), f^*(u_k, l_{i,k})\}$ units of flow from both $(l_{i,k}, u_k)$ and $(u_k, l_{i,k})$. By performing these updates one obtains a loop-free flow f^* that is nested.

4.3 Constructing an Integral Flow

We describe the three main flow modification steps.

4.3.1 *Step 1: Flow scaling.* Let f^* be the minimum-cost, loop-free flow. Algorithm $\mathcal{A}_2$ multiplies f^* by a factor of τ on all edges of the network. At the same time it multiplies all edge capacities and supplies/demands by τ. Then it deletes the flow of commodity 2 and the supplies/demands at a_k and b_k, $1 \leq k \leq n-1$. The resulting flow f^1 of commodity 1 is feasible. Additionally, in each component it is nested and loop-free. There holds $cost(f^1) \leq \tau\, cost(f^*)$.

In Steps 2 and 3 flow f^1 is modified. As indicated above, the modification are executed independently for the components. Therefore, in the description of Steps 2 and 3 we concentrate on one component C that ships τm_c units of flow. The flow modifications never increase the cost. At all times the flow remains nested and loop-free.

4.3.2 *Step 2: Rounding flow on the upper path.* Given f^1, $\mathcal{A}_2$ rounds it so that the flow becomes integral on the upper path of C. Along edge (u_k, u_{k+1}) the amount of flow will be $\min\{m_c, \lfloor \tau f^*(u_k, u_{k+1}) \rfloor\}$. Recall that m_c is the number of servers in the class represented by C. We first describe how to reduce f^1 so that on any edge (u_k, u_{k+1}) the flow is $\lfloor \tau f^*(u_k, u_{k+1}) \rfloor$. $\mathcal{A}_2$ makes four passes over C. First, in Step 2.1, it rounds valleys with low flow. Then, in Steps 2.2 and 2.3, it modifies flow on edge sequences with increasing and decreasing flow, respectively. Finally, in Step 2.4, it takes care of flow peaks. Given this flow, we further reduce it so that the flow on any edge of the upper path does not exceed m_c. At any time, for a current flow f, we say that the flow *increases at* u_k if $k = 1$ and $f(u_1, u_2) > 0$ or if $1 < k < n$ and $f(u_{k-1}, u_k) < f(u_k, u_{k+1})$. Similarly, the flow *decreases at* u_k if $1 < k < n$ and $f(u_{k-1}, u_k) > f(u_k, u_{k+1})$ of if $k = n$ and $f(u_{n-1}, u_n) > 0$. Initially in Step 2, let $f = f^1$.

Step 2.1: Valleys. A *valley* is a path $P(k, k')$, $1 < k < k' < n$, on the upper path of C such that the flow decreases at u_k, increases at $u_{k'}$ and is constant for all the edges of $P(k, k')$. Formally, the last condition indicates that $f(e) = f(u_k, u_{k+1})$, for all edges e of $P(k, k')$. $\mathcal{A}_2$ scans C. Whenever it encounters a valley $P(k, k')$ with a non-integral amount of flow, it invokes the following procedure that reduces the flow to $\lfloor f(u_k, u_{k+1}) \rfloor$.

Flow update procedure: For the given valley $P(k, k')$, the flow decreases at u_k. The procedure determines the smallest j such that flow is routed from u_k to $l_{j,k}$ and shipped on lower path j. In the full paper we prove that $P_j(k, k')$ routes flow. Hence, as $P_j(k, k')$ routes flow, in the unscaled minimum-cost flow f^* path $P_j(k, k')$ also ships flow. Routing the flow on the upper path would have been a feasible option as well. This implies that the total edge cost of $P_j(k, k')$ is upper bounded by that of $P(k, k')$. The procedure updates the flow as follows. It remove $\delta = f(u_k, u_{k+1}) - \lfloor f(u_k, u_{k+1}) \rfloor$ units of flow from $P(k, k')$ and instead routes them along $P_j(k, k')$. This does not increase the cost. The resulting flow in C remains nested. Modifying all valleys takes $O(n\sigma)$ time.

Step 2.2: Flow increases. In a second pass over C algorithm $\mathcal{A}_2$ identifies vertices u_k at which the flow increases. If $f(u_k, u_{k+1})$

is not integral, it is reduced to $\lfloor f(u_k, u_{k+1})\rfloor$. Starting at u_1 or at a vertex representing the end of a valley, $\mathcal{A}_2$ performs a sequence of vertex inspections and possible flow updates. The sequence ends at a vertex at which the flow decreases. The algorithm then searches for the end of the next valley and continues.

Formally, let $u_{k'}$ be a vertex such that $k' = 1$ or $u_{k'}$ is the last vertex of a valley. When located at $u_{k'}$, $\mathcal{A}_2$ determines the smallest k'' with $k'' > k'$ such that the flow decreases at $u_{k''}$. The algorithm inspects the vertices u_k, $k' \le k < k'' - 1$, in order of increasing index. If $f(u_k, u_{k+1})$ is not integral, the procedure described in the next paragraph is invoked, which reduces the flow to $\lfloor f(u_k, u_{k+1})\rfloor$. When the procedure is executed at u_k, there holds $u_k = u_1$ or the flow $f(u_{k-1}, u_k)$ on the preceding edge is integral. The latter condition holds true because if $u_k = u_{k'}$ is the last vertex of a valley, then the flow along the incoming edge has been made integral in Step 2.1. $\mathcal{A}_2$ considers vertices in order of increasing index, starting at $u_{k'}$. When u_k, $k' < k < k'' - 1$, is inspected, the flow on the edges between $u_{k'}$ and u_k is already integral.

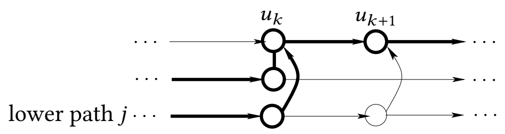

Figure 5: A flow increase at u_k.

Flow update procedure: The procedure updates flow at a vertex u_k where the flow increases and $f(u_k, u_{k+1})$ is not integral. Let again $\delta = f(u_k, u_{k+1}) - \lfloor f(u_k, u_{k+1})\rfloor$. Either $k = 1$ or the flow on (u_{k-1}, u_k) is integral. Hence at least δ units of flow are shipped from lower paths into u_k. While $\delta > 0$, the procedure executes the following steps. Let j be the largest integer such that the flow from $l_{j,k}$ to u_k is positive. Figure 5 depicts the general flow configuration. Let $\delta_j = f(l_{j,k}, u_k)$ and $\delta' = \min\{\delta, \delta_j\}$. The procedure reduces flow on $(l_{j,k}, u_k)$ and (u_k, u_{k+1}) by δ units. Instead it ships δ units of flow from $l_{j,k}$ to u_{k+1} along lower path j, i.e. via $l^a_{j,k}$ and $l_{j,k+1}$. Then δ is reduced by δ'. The flow update decreases the cost of the flow by $(r_0 - r_j)(t_{k+1} - t_k) > 0$. Here r_0 and r_j are the cost coefficients along the upper path and lower path j, respectively. More precisely, edge (u_k, u_{k+1}) has a cost of $r_0(t_{k+1} - t_k)$ and $(l_{j,k}, l^a_{j,k})$ has a cost of $r_j(t_{k+1} - t_k)$. The modified flow remains nested; detailed arguments are given in the full paper. The running time of one execution of the procedure is $O(\sigma)$. The running time of the entire pass over C is $O(n\sigma)$.

Step 2.3: Flow decreases. The flow modifications are symmetric to those described in Step 2.2. Algorithm $\mathcal{A}_2$ makes another pass over C, this time from right to left starting at u_n. It searches for vertices u_k at which the flow decreases. If the flow $f(u_{k-1}, u_k)$ on the incoming edge is not integral, then it is reduced to $\lfloor f(u_{k-1}, u_k)\rfloor$. A detailed description is provided in the full version of the paper.

Step 2.4: Peaks. A *peak* is an edge (u_k, u_{k+1}) such that the flow increases at u_k and decreases at u_{k+1}, see also Figure 6. After $\mathcal{A}_2$ has executed Steps 2.1–2.3, the only edges on the upper path with a non-integral amount of flow are peaks. Algorithm $\mathcal{A}_2$ traverses C. For each peak (u_k, u_{k+1}) with a non-integral amount of flow, it invokes the following routine.

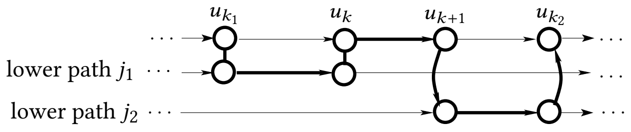

Figure 6: A flow peak on (u_k, u_{k+1}).

Flow update procedure: Again $\delta = f(u_k, u_{k+1}) - \lfloor f(u_k, u_{k+1})\rfloor$. Let j_1 be the largest integer such that $f(l_{j_1,k}, u_k) > 0$, i.e. flow is routed from lower path j_1 to u_k. Let j_2 be the largest integer such that $f(u_{k+1}, l_{j_2,k+1}) > 0$. There are two basic cases.

If $j_1 = j_2$, then let $\delta_1 = f(l_{j_1,k}, u_k)$ and $\delta_2 = f(u_{k+1}, l_{j_2,k+1})$. Furthermore, let $\delta' = \min\{\delta, \delta_1, \delta_2\}$. The procedure removes δ' units of flow from the path connecting $l_{j_1,k}$ and $l_{j_1,k+1}$ along the upper path. Specifically, it removes δ' flow units from the edges $(l_{j_1,k}, u_k)$, (u_k, u_{k+1}) and $(u_{k+1}, l_{j_1,k+1})$. Instead it sends δ' units of flow from $l_{j_1,k}$ to $l_{j_1,k+1}$ via $l^a_{j_1,k}$ on lower path j_1. The reduction in the cost of the flow is $\delta(r_0 - r_{j_1})(t_{k+1} - t_k) + \delta\Delta_{j_1} > 0$. Here Δ_j is the cost of $(l_{j,k}, u_k)$, for any $1 \le j \le \sigma$ and $1 \le k < n - 1$.

If $j_1 \ne j_2$, then let k_1 be the largest integer such that $P_{j_1}(k_1, k)$ routes flow. Let δ_1 be the largest value such that every edge of $P_{j_1}(k_1, k)$ routes at least δ_1 units of flow. Similarly, let k_2 be the smallest integer such that $P_{j_2}(k+1, k_2)$ routes flow. Let δ_2 be the largest value such that every edge of $P_{j_2}(k+1, k_2)$ routes at least δ_2 units of flow. Figure 6 shows the case that $j_1 < j_2$. Let $\delta' = \min\{\delta, \delta_1, \delta_2\}$. The procedure removes δ' units of flow from $P_{j_1}(k_1, k)$, edge (u_k, u_{k+1}) and $P_{j_2}(k+1, k_2)$. If $j_1 < j_2$, it instead sends these δ' units on path $P_{j_2}(k_1, k_2)$. If $j_1 > j_2$, it routes the δ' units along $P_{j_1}(k_1, k_2)$. Thus, in any case the deeper low-power state is used. It is not hard to verify that the cost of the flow decreases.

In any case δ is reduced by δ'. In the full paper we prove that the new flow is nested. One call of the procedure takes $O(n^2\sigma)$; the rounding of all peaks can be accomplished in $O(n^3\sigma)$ time.

Step 2.5: Flow reduction to m_c. It remains to reduce the flow to m_c on edges (u_k, u_{k+1}) where the flow after Steps 2.1–2.4 is higher. This can be done using the procedures that handle flow increases and peaks. Details are given in the full paper.

4.3.3 Step 3: Packing flow on the lower paths. Given the flow f^2 constructed in Step 2, $\mathcal{A}_2$ packs flow on the lower paths of the considered component C so that the final flow becomes integral. During the modification the flow on the upper path of C does not change. Moreover, the cost of the flow will not increase.

Auxiliary edges. In order to separate flow that has already been made integral from the original one, we need auxiliary edges. For every edge e in C, except for those in the upper path, we add an auxiliary edge e'. More precisely, for every link $e = (v, w)$ not contained on the upper path, there is the original edge and a new auxiliary edge.

Initially, the flow f^2 is routed on the upper path and the original edges of the lower paths. In a series of rounds $\mathcal{A}_2$ removes flow from the original edges, packs it and adds it to the auxiliary edges. On the auxiliary edges, the flow is always integral. The process ends when there is no flow on the original edges. Then the original edges are removed so that, for each edge, there is only one copy.

We observe that since f^2 is integral on the upper path and loop-free, only integral amounts of flow enter/leave the upper path from/to the lower paths. This invariant will be maintained at all times during the flow transformation. For every edge, the total flow on the original and auxiliary copy will always be at most τm_c.

Matching pairs. The flow packing procedure works with with the notion of a *matching pair*. Such a pair consists of two vertices u_k and $u_{k'}$, $1 < k < k' < n$, with the following properties: (a) Flow is routed from u_k to the lower paths on original edges $(u_k, l_{j,k})$, $1 \le j \le \sigma$; (b) flow is routed into $u_{k'}$ from lower paths on original edges $(l_{j,k'}, u_{k'})$, $1 \le j \le \sigma$; (c) there exists no vertex $u_{k''}$ with $k < k'' < k'$ that satisfies (a) or (b). While there exists a matching pair $\mathcal{A}_2$ executes the following flow packing routine. Unless otherwise stated, $f(e)$ refers to the current flow on the original copy of e.

Packing procedure. Let u_k and $u_{k'}$ be the given matching pair. Let δ_k be the total amount of flow routed from u_k to lower paths on original edges $(u_k, l_{j,k})$, $1 \le j \le \sigma$. Similarly, let $\delta_{k'}$ be the total amount of flow shipped into $u_{k'}$ from lower paths along original edges $(l_{j,k'}, u_{k'})$, $1 \le j \le \sigma$. Both δ_k and $\delta_{k'}$ are integral. If $\delta_k \le \delta_{k'}$, then let J be the set of integers j with $1 \le j \le \sigma$ such that $f(u_k, l_{j,k}) > 0$. Define $\delta_j = f(u_k, l_{j,k})$, for any $j \in J$. There holds $\sum_{j \in J} \delta_j = \delta_k$. If $\delta_k > \delta_{k'}$, then let J be the set of integers j with $1 \le j \le \sigma$ such that $f(l_{j,k'}, u_{k'}) > 0$. Define $\delta_j = f(l_{j,k'}, u_{k'})$, for any $j \in J$. There holds $\sum_{j \in J} \delta_j = \delta_{k'}$. In any case, for $j \in J$, consider the path $P_j(k, k')$. In the full paper we formally prove that $P_j(k, k')$ routes δ_j flow units, for any $j \in J$.

The procedure for packing flow determines the integer $j' \in J$ such that the total edge cost of $P_{j'}(k, k')$ is minimal among $P_j(k, k')$ with $j \in J$. Then, for every $j \in J$, it removes δ_j units of flow from the original edges of $P_j(k, k')$. Finally, it routes $\min\{\delta_k, \delta_{k'}\}$ units of flow on the new edges of $P_{j'}(k, k')$. The new flow remains nested because an already existing routing path with positive flow is selected. The cost does not increase, due to the choice of j'.

Every time the procedure is invoked for a matching pair u_k and $u_{k'}$, the flow leaving u_k or entering $u_{k'}$ on original edges drops to 0. Thus all executions of the procedure take $O(n^2\sigma)$ time. When there exists no matching pair anymore, the remaining flow on original edges along paths $P_\sigma(1, k')$ and $P_\sigma(k, n)$ can be transferred without modification to the auxiliary edges.

4.4 Construction of the Schedule

Let f^3 denote the flow obtained in Step 3.

Lemma 4.6. *Flow f^3 corresponds to a schedule Σ with τm_i servers of type i, $1 \le i \le \tau$. In $[t_k, t_{k+1})$ exactly $d_{i,k} = \min\{m_i, \lfloor \tau f^*(u_{i,k}, u_{i,k+1}) \rfloor\}$ servers of type i are in the active state, $1 \le i \le \tau$ and $1 \le k \le n-1$. The energy consumed by the servers of type i is equal the cost of f^3 in C_i, $1 \le i \le \tau$.*

Lemma 4.7. *Let Σ_i be a schedule for τm_i servers of type i in which exactly $d_{i,k}$ servers are in the active state during $[t_k, t_{k+1})$, where $d_{i,k} \le m_i$ and $1 \le k \le n-1$. Then there exists a schedule Σ_i' for m_i servers of type i in which the servers numbered 1 to $d_{i,k}$ are in the active state during $[t_k, t_{k+1})$. The energy consumed by Σ_i' is upper bounded by that of Σ_i.*

Given the integral flow f^3, algorithm $\mathcal{A}_2$ constructs a feasible schedule Σ^* for $\mathcal{I}$. For each server type $\mathcal{S}_i$, $1 \le i \le \tau$, $\mathcal{A}_2$ builds an optimal schedule Σ_i^* such that $d_{i,k}$ of the m_i servers in $\mathcal{S}_i$ are in the active state during $[t_k, t_{k+1})$, $1 \le k \le n-1$. These schedules $\Sigma_1^*, \ldots, \Sigma_\tau^*$ are then combined to form Σ^*. Consider any i with $1 \le i \le \tau$. In a first step, by Lemma 4.7, Σ_i^* just specifies that the servers numbered 1 to $d_{i,k}$ are in the active state during $[t_k, t_{k+1})$, for any $1 \le k \le n-1$. Then, while a server is not required to be active according to the specification, $\mathcal{A}_2$ selects an optimal state. Suppose that at time t_k the number of required servers decreases, i.e. $d_{i,k-1} > d_{i,k}$. Algorithm $\mathcal{A}_2$ determines states for $d_{i,k-1} - d_{i,k}$ servers that may power down. This is done as follows. Initially, $\mu := d_{i,k}$. While $\mu < d_{i,k-1}$, $\mathcal{A}_2$ finds the next time $t_{k'}$ such that $d_{i,k'} \ge \mu + 1$, i.e. at least $\mu + 1$ servers are active. It chooses an optimal state to be assumed by servers numbered $\mu + 1, \ldots, \min\{d_{i,k-1}, d_{i,k'}\}$ at time t_k. This is the state s_{i,j^*} with $j^* = \arg\min_{1 \le j \le \sigma_i}\{r_{i,j}(t_{k'} - t_k) + \Delta_{i,j}\}$. Then $\mu := \min\{d_{i,k-1}, d_{i,k'}\}$.

By Lemmas 4.6 and 4.7, the energy consumed by Σ_i^* is upper bounded by the cost incurred by f^3 in component C_i. Thus the energy consumed by the combined schedule Σ^* is at most $cost(f^3) \le \tau cost(f^*)$. Schedule Σ^* is feasible because, by Lemma 4.4, $\sum_{i=1}^{\tau} d_{i,k} \ge d_k$. The proof of Theorem 4.1 is complete.

REFERENCES

[1] The Advanced Configuration and Power Interface. The latest specification 6.1 (January 2016) is available e.g. at UEFI.org.
[2] R.J. Ahuja, T.L. Magnanti and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.
[3] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report No. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, 2009.
[4] J. Augustine, S. Irani and C. Swamy. Optimal power-down strategies. *SIAM J. Comput.*, 37(5):1499–1516, 2008.
[5] Y. Azar, Y. Bartal, E. Feuerstein, A. Fiat, S. Leonardi and A. Rosén. On capital investment. *Algorithmica*, 25(1):22–36, 1999.
[6] L.A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.
[7] M. Dayarathna, Y. Wen and R. Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys and Tutorials*, 18(1):732–794, 2016.
[8] G. Fettweis and E. Zimmermann. ICT energy consumption – trends and challenges. *Proc. 11th International Symp. on Wireless Personal Multimedia Communications*, 2008.
[9] A. Gandhi and M. Harchol-Balter. How data center size impacts the effectiveness of dynamic power management. *49th Annual Allerton Conference*, 1164–1169, 2011.
[10] A. Gandhi, M. Harchol-Balter and I.J.B.F. Adan. Server farms with setup costs. *Perform. Eval.*, 67(11):1123–1138, 2010.
[11] Z.J. Haas and S. Gu. On power management policies for data centers. *Proc. IEEE Int. Conf. on Data Science and Data Intensive Systems (DSDIS)*, 404–411, 2015.
[12] W. van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet and P. Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, 2014.
[13] Heterogeneous computing. `https://en.wikipedia.org/wiki/Heterogeneous_computing`
[14] S. Irani, S.K. Shukla and R.K. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Trans. Embedded Comput. Syst.*, 2(3):325–346, 2003.
[15] A.R. Karlin, C. Kenyon and D. Randall. Dynamic TCP acknowledgment and other stories about e/(e-1). *Algorithmica*, 36(3):209–224, 2003.
[16] A.R. Karlin, M.S. Manasse, L. Rudolph and D.D. Sleator. Competitive snoopy caching. *Algorithmica* 3:77–119, 1988.
[17] S. Khuller, J. Li and B. Saha. Energy efficient scheduling via partial shutdown. *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1360–1372, 2010.
[18] J. Li and S. Khuller. Generalized machine activation problems. *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 80–94, 2011.
[19] A. Levi and B. Patt-Shamir. Non-additive two-option ski rental. *Theor. Comput. Sci.*, 584:42–52, 2015.
[20] M. Lin, A. Wierman, L.L.H. Andrew and E. Thereska. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Trans. Netw.*, 21(5):1378–1391, 2013.
[21] Z. Lotker, B. Patt-Shamir and D. Rawitz. Rent, lease, or buy: Randomized algorithms for multislope ski rental. *SIAM J. Discrete Math.* 26(2):718–736, 2012.
[22] K. Wang, M. Lin, F. Ciucu, A. Wierman and C. Lin. Characterizing the impact of the workload on the value of dynamic resizing in data centers. *Proc. IEEE INFOCOM*, 515–519, 2013.