Resource Allocation Algorithms for Virtualized Service Hosting Platforms

Mark Stillwell[a], David Schanzenbach[a], Frédéric Vivien[b], Henri Casanova[*,a]

[a]*Department of Information and Computer Sciences*
University of Hawai'i at Mānoa, USA.
[b]*INRIA, Université de Lyon, LIP,*
UMR 5668 ENS-CNRS-INRIA-UCBL, Lyon, France

Abstract

Commodity clusters are used routinely for deploying service hosting platforms. Due to hardware and operation costs, clusters need to be shared among multiple services. Crucial for enabling such *shared hosting platforms* is virtual machine (VM) technology, which allows consolidation of hardware resources. A key challenge, however, is to make appropriate decisions when allocating hardware resources to service instances. In this work we propose a formulation of the resource allocation problem in shared hosting platforms for static workloads with servers that provide multiple types of resources. Our formulation supports a mix of best-effort and QoS scenarios, and, via a precisely defined objective function, promotes performance, fairness, and cluster utilization. Further, this formulation makes it possible to compute a bound on the optimal resource allocation. We propose several classes of resource allocation algorithms, which we evaluate in simulation. We are able to identify an algorithm that achieves average performance close to the optimal across many experimental scenarios. Furthermore, this algorithm runs in only a few seconds for large platforms and thus is usable in practice.

Key words: Resource Allocation, Virtual Machine, Cluster

[*]Corresponding Author
Email address: `henric@hawaii.edu` (Henri Casanova)

1. Introduction

In *service hosting*, clustered servers run components of continuously running services, such as Web and e-commerce applications. High utilization is paramount for justifying cluster costs [1]. While running services on dedicated servers achieves performance isolation, it leads to low utilization. There is thus a strong incentive for sharing cluster resources among services, establishing the so-called *shared hosting platforms* [61]. The challenge is then to allocate appropriate resource shares to services that have conflicting resource demands. The term "appropriate" can take various and non-exclusive meanings: meeting QoS requirements, ensuring fairness among services, increasing platform, maximizing platform utilization, maximizing service-defined utility functions, etc.

In this paper we frame the resource allocation problem for shared hosting platform as a constrained optimization problem, for which efficient algorithms can be developed. We make the following contributions: (i) We define the resource allocation problem for a static workload of services that are each fully contained in a single VM instance; this definition accounts for multiple resource dimensions, supports a mix of best-effort and QoS scenarios, and promotes performance, fairness, and high resource utilization (Section 3); (ii) We establish the complexity of the problem and give a mixed integer linear program formulation that leads to a bound on the optimum (Section 3); (iii) We propose several algorithms to solve the problem (Section 4); (iv) We evaluate these algorithms in simulation (Section 5); and (v) we discuss how our approach can be extended to services that comprise multiple VM instances (Section 6).

2. Related Work

Several systems have been proposed to manage resources in shared hosting platforms [51, 47, 29, 39, 10, 53, 62, 69, 25, 43, 3, 70]. Our work provides algorithmic solutions that can be implemented in these systems. Many of these works do not emphasize the development of solid resource allocation algorithms, but rather focus on the challenges of system implementation, often using naïve resource allocation strategies as place-holders. Other works attempt to target the resource allocation problem in its full complexity via intricate and multi-faceted engineering solutions. Instead, we take a step back and attempt to formulate a precise optimization problem that makes a few key assumptions. Good algorithms for solving this problem then provide sound bases on which to address the problem in its full complexity.

[61] attempts to place applications onto a shared cluster using intelligent "overbooking", i.e., sharing servers among application instances. Both this work and ours account for application QoS requirements, but the approach in [61] attempts to maximize resource provider revenue while we focus on application performance and fairness in terms of resource shares. A key contribution in [61] is the validation of a number of application profiling techniques for obtaining statistical bounds on resource usage and minimum resource needs. We build on these results and assume that such bounds are known to our algorithms. Our

key contribution is that we provide a formal definition of the resource allocation problem for a clearly defined objective function. We can therefore propose algorithms inspired by the theoretical literature that go beyond the common sense but simple heuristics used in [61]. The authors acknowledge that it is difficult to consider multiple resource dimensions simultaneously (e.g., CPU rate, network bandwidth, RAM space, and disk space). Thanks to our problem formulation our approach is applicable to an arbitrary number of resource dimensions.

[3] formulates the resource allocation problem as a constrained optimization problem, where each service is characterized by its desired resource share. One difference with our work is that we explicitly consider multiple resource dimensions while [3] consider a server as a monolithic resource. Also, the approach in [3] first optimizes a linear objective function, namely, the average deviation between a service's resource share and its desired resource share. This can lead to unfair schedules and [3] uses a second optimization step, this time with a quadratic objective function, that includes a bias term to improve fairness. By contrast, we use a linear objective function that naturally captures fairness, as inspired by the theoretical job scheduling literature [6, 35].

[53] proposes a resource allocation scheme for service hosting platforms, based on the notion of optimizing average service "yield", where the yield is defined based on generic utility functions of service response times. Similarly, [45] proposes a technique that relies on queuing models of services and attempts to maximize global utility based on a dynamic programming approach. In this work we do not consider utilities, but, instead, consider solely the resource share allocated to a service as the sole performance metric. This metric is generic, correlated to popular metrics of performance such as service response time, directly allows the specification of minimum resource shares, and allows us to formalize and provide algorithmic solutions to the resource allocation problem.

This work focuses on the static resource allocation problem, i.e., assuming that the workload does not change. Resource allocations can then be recomputed reactively when the workload changes, provided our resource allocation algorithms run quickly. Several authors have studied the dynamic resource allocation problem directly. In this case, given the complexity of the problem, even with a single resource dimension authors resort to simple (typically greedy) heuristics [53, 17, 60, 26, 21]. An important concern for resource allocations adaptation is to minimize overhead. To this end, [2] attempts to compute static resource allocations that should delay as much as possible the need for adaptation in the face of dynamic workloads. Other works attempt to minimize the change in resource allocations due to adaptation [22, 32, 9].

The challenges of resource allocation and of application modeling for multi-tier applications are studied in [60] and [40]. Going further in application modeling complexity, [2] proposes resource allocation algorithms to meet QoS requirements of continuous applications structured as Directed Acyclic Graphs on a heterogeneous platform. For most of this work we assume that a service is fully encapsulated in a VM instance, and thus do not account for interactions between service components. Nevertheless, in Section 6 we discuss a limited multi-tier scenario in which each tier is in a separate VM instance.

Due to our precise formulation of the problem as a linear program, we are able to quantify how far our algorithms are from optimal. Among the aforementioned works, only [2] provides a loose bound on the optimal. [7, 48, 21] study static and dynamic resource allocation problems with the objective of minimizing the number of servers (or the server cost). While our target problem is different, due to a different objective and to different models of resource needs of services, those works also provide linear program formulations. [7] uses such a formulation to compute optimal solutions for small problem instances.

Like most resource allocation problems, our problem is related to bin packing. However, it is related to a variant in which bins and items are multi-dimensional vectors, or *vector packing*. Versions of standard greedy algorithms (First Fit, Best Fit, Worst Fit, Next Fit, etc.) have been proposed for vector packing [33, 38], their worst-case behaviors have been studied [33, 16], as well as their average behavior over practical instances [38, 50]. Results show that these algorithms have performance guarantees $d+\delta$, where d is the number of resource dimensions and δ is some constant ≤ 1. An algorithm with performance guarantee $d+\varepsilon$, for any $\varepsilon > 0$, can be obtained by reusing the $(1+\varepsilon)$-guaranteed bin packing algorithm in [18]. This guarantee was improved in [11], which proposes an algorithm with a $O(\ln d)$ guarantee (the guarantee approaches $2 + \ln d$ for large d). More recently, [4] gave a complex algorithm with a $1 + \ln d$ guarantee. Vector packing heuristics beyond the standard greedy ones are proposed in [36] and in [38]. These heuristics do not provide tight performance guarantees but may exhibit good average-case behavior. Genetic algorithms have also been used to solve vector packing problems that arise naturally from resource allocation problems [48, 24, 21]. In this work we consider all standard greedy heuristics, the guaranteed algorithm in [11], the heuristics from [36] and [38], and a genetic algorithm similar to the one used in [48].

3. Problem Definition

3.1. System Model

We consider a shared hosting platform that comprises homogeneous servers clustered with a high-speed switched interconnect. A resource allocation system controls how multiple services share the platform. Each service consist of one or multiple VM instances and the system ensures that requests to the services are dispatched to appropriate servers. The system imposes precise resource shares for each VM instance via VM monitors [5, 63, 41], and can migrate VM instances among the servers. In this work we assume that VM technology allows precise sharing of hardware resources among VM instances. This is certainly possible today for CPU, RAM space, and disk space. For instance, it has been shown that sharing and performance isolation for CPU resources is low-overhead, accurate within 1%, and rapidly adaptable [52]. Furthermore, sharing can be arbitrary. For instance, the Xen Credit CPU scheduler can allow 3 Virtual Machine instances to each receive 33.3% of the CPU capacity of a dual-core machine [27]. Virtualization of other resources, such as I/O resources, is more challenging [13, 65, 66] but is an active area of research [68, 44]. Recent works also target the virtualization of full memory hierarchy (buses and caches) [42].

3.2. Resource and Service Models

Each server provides several resources (e.g., CPU, RAM space, I/O bandwidth, disk space). For now we assume that each service consists of a single VM instance, and consider multi-VM services in Section 6. We also assume a static workload, meaning that each service has constant resource needs.

We consider two kinds of resource needs: *rigid* and *fluid* (the corresponding terminology in [32] is "load-independent" and "load-dependent"). A rigid need denotes that a specific fraction of a resource is required. The service cannot benefit from a larger fraction and cannot operate with a smaller fraction. A fluid need specifies the maximum fraction of a resource that the service could use if alone on the server. The service cannot benefit from a larger fraction, but can operate with a smaller fraction at the cost of reduced performance. For instance, a service could have two rigid needs: it could require 50% of a server's RAM and 20% of a server's disk space. The service could have two fluid needs: it could use up to 40% of a server's I/O bandwidth and up to 60% of a server's CPU. In this example, the service cannot use both resources fully, for instance because of interdependence between I/O and computation. For each fluid resource need we can define the ratio between the resource fraction allocated and the maximum resource fraction potentially used. We call this ratio the *yield* of the fluid resource need. For instance, if a service has a fluid CPU need of 60% but is allocated only 42% of the CPU, then the yield is $42/60 = 0.7$.

We assume that, within a service, the utilizations of all resources corresponding to fluid needs are linearly correlated. For the previous example, if the service were to be allocated only 20% of the I/O bandwidth (i.e., half of what it could potentially use), then it would use only 30% of the CPU (i.e., also half of what it could potentially use). This assumption is easily justifiable for many services, and in particular Internet services that process incoming user requests. In this case, some percentage of a resource, e.g., the CPU, is used to process requests at a given rate. If CPU consumption is throttled down to, say, half of this percentage, then requests are processed at half the original rate. As a result, consumption for other resources are also halved. More generally, the yields of all fluid resource needs are identical, and we call their value simply the yield of the service. Our approach can be extended to the case in which the linear correlation assumption does not hold. In this case, the relationship between the utilizations of resources corresponding to fluid needs must be described by a function that, for a given yield, returns the resource fractions corresponding to fluid needs so that this yield can be achieved. The yield is then no longer a resource fraction, but a fraction of the maximum achievable application "speed", and the function is simply an application performance model. The vector packing algorithms presented in Section 4.5, which outperform all other algorithms in this work, can be used directly if such a performance model is available.

With the above definition, the yield of a service takes values between 0 and 1, with 0 corresponding to the case in which the service is not allocated any resource. However, there could be a lower bound on the yield of a service, which specifies a QoS requirement. For the earlier example, the yield could be constrained to be higher than 0.4, which means that the CPU fraction allocated to the job would be at least $0.4 \times 60\% = 24\%$. Some services may not have any QoS requirement and resources can be allocated to them in a best-effort fashion. We use the term *constrained fluid need* to denote a service's fluid need multiplied by the service's QoS requirement. If the QoS requirement is 0, then the constrained fluid need is 0. While not true in all cases [17], for simplicity we assume that rigid resource needs are completely independent from fluid resource needs (and from each other).

3.3. Determining Resource Needs

The previous section defines service needs in terms of resource fractions. An important question is how to determine actual values of these resource fractions. Beyond users specifying resource fractions directly, one approach is to rely on service benchmarking [61]. This is reasonable given that a shared hosting platform often hosts a moderate number of well-known services over long periods of time. It is also possible to build analytical models of resource needs and of their temporal trends [60, 23, 49, 17], and even to augment these models to account for virtualization overheads [67]. Finally, another approach consists in monitoring services as they run to determine their needs [70, 53, 45, 32, 9]. [25] uses a framework dedicated to such resource need discovery [54]. VM instances monitoring can be based on facilities such as the XenMon facility [28]. VM instance resource needs can also be discovered via a combination of introspection and configuration variation [30, 31].

In this work we reason on resource fractions allocated to services and optimize a metric, the yield, which is computed based on these fractions. In practice, however, resource management objectives are expressed based on higher level metrics that are related to user concerns, such as service response time or throughput. This work relies on the fact that these higher level metrics are directly related to resource fractions allocated to services. This observation has been made repeatedly in the literature and several models that link resource shares to response time and or throughput have been developed [12, 34, 9, 46, 56, 64, 55]. For instance, in [12], the authors model response time as a function of CPU and memory resource fractions allocated to services. These models are validated for real-world services (Tomcat, MySQL). In [34] a model of response time as a function of allocated CPU fraction is developed for a Web application benchmark (RUBBoS), using linear interpolation. Similar models for response time and for throughput are proposed and validated in [46] for two multi-tier application benchmarks (RUBiS and TPC-W). A response time model for TPC-W is also proposed in [55], while [9] proposes a response time model for several synthetic applications. All aforementioned works handle only services with QoS requirements. Our approach can handle a broader scenario, allocating remaining resources to services beyond their QoS requirements and to best-effort services that do not have any QoS requirements, in a way that improves overall performance and fairness. We conclude that reasoning on resource shares directly, in our case via the yield metric, is a good way to achieve higher-level resource management objectives. Since our approach is agnostic to these objectives it can be applied in a variety of contexts. As an example, in [59] it is shown that yield optimization is effective for parallel job scheduling.

3.4. Objective and Constraints

Recall that the yield of a service takes values between a specified minimum positive value and 1. The case in which the yield of a service is below its minimum value corresponds to a failure of the resource allocation procedure. Informally, the yield quantifies the "happiness" of a service, i.e., how close it is to receiving the maximum amount of resources it can use. To compare yields across services with various minimum yield requirements, we define the *scaled yield* of a service as follows:

$$\text{scaled yield} = \frac{\text{yield} - \text{minimum yield}}{1 - \text{minimum yield}} .$$

Consider two services with minimum yield requirements of 0.2 and 0.4, respectively. Then a yield of 0.8 for the first service and of 0.85 for the second achieve an identical scaled yield for both services, of 0.75. In other terms, each service experiences a yield that is 75% of the way between its minimum and maximum yield values. For a best-effort service the minimum yield is 0, and the scaled yield is equal to the yield. The above equation is undefined for a service whose minimum yield is equal to 1. Such a service is always "happy" in a valid resource allocation, and we defined its scaled yield as 1 in a valid allocation and 0 otherwise. If the scaled yield of a service is negative then resource allocation fails. Resource allocation also fails if a rigid need of a service cannot be met. In all that follows, we use the term "yield" to mean "scaled yield", while we sometimes explicitly refer to the "unscaled yield."

We formulate the resource allocation problem, ResAlloc, as maximizing the minimum yield over all services. Informally, we wish to make the least happy service as happy as possible, in a view to promoting both fairness and performance. Our yield metric is related to a popular metric in the job scheduling literature: the *stretch* [6]. The stretch is applicable to time-bound computational jobs rather than continuous services. It is defined as the job's flow time, i.e., time between job submission and job completion, divided by the flow time that would have been achieved had the job been alone in the system. Minimizing the maximum stretch has been recognized as a way to optimize average flow time while ensuring that jobs do not experience high relative flow times. Consequently, it is a way to optimize *both* performance and fairness, while simply minimizing average stretch is prone to starvation [6, 35]. In the same spirit, we aim at maximizing the minimum yield.

Maximizing the minimum yield is subject to a few constraints. The first constraint is that the resource capacities of the servers not be overcome. Another constraint is that a service, which is encapsulated inside a single VM instance, be allocated to a single server. This implies that there is no service migration. While this may seem natural given that we consider a static workload, it turns out that migration could be used to achieve better minimum yield. Assuming that migration can be done with no overhead or cost whatsoever, as often done in the theoretical literature, migrating jobs among servers periodically can increase the minimum yield [57]. Unfortunately, the assumption that migration comes at no cost or overhead is not realistic. While VM migration can be low-latency [14], it consumes network resources, and one should bound the number of migrations. We leave the use of periodic migrations outside the scope of this work.

3.5. Linear Program Formulation

We formulate ResAlloc as a Mixed Integer Linear Program (MILP), i.e., a linear program with rational and integer variables. We consider $N > 0$ services, indexed by $i = 1, \ldots, N$. The cluster comprises $H > 0$ identical physical servers, indexed by $h = 1, \ldots, H$. Each server provides d types of resources, indexed by $j = 1, \ldots, d$. Fractions of these resources can be allocated to services. For each service i, r_{ij} denotes its resource need for resource type j, as a resource fraction between 0 and 1. δ_{ij} is a binary value that is 1 if r_{ij} is a rigid need, and 0 if r_{ij} is a fluid need. We use $\hat{y}_i$ to denote the minimum yield requirement of service i, a value between 0 and 1.

We can now define the variables of our linear program. We define a binary variable e_{ih} that is 1 if service i runs on server h and 0 otherwise. We denote by y_{ih} the unscaled yield of service i on server h, which must be equal to 0 if the service does not run on the server. With these definitions the constraints of our linear program are as follows, with Y denoting the minimum yield:

$$\begin{aligned}
\forall i, h \quad & e_{ih} \in \{0, 1\}\ , \quad y_{ih} \in \mathbb{Q} && (1)\\
\forall i \quad & \textstyle\sum_h e_{ih} = 1 && (2)\\
\forall i, h \quad & 0 \leq y_{ih} \leq e_{ih} && (3)\\
\forall i \quad & \textstyle\sum_h y_{ih} \geq \hat{y}_i && (4)\\
\forall h, j \quad & \textstyle\sum_i r_{ij}(y_{ih}(1 - \delta_{ij}) + e_{ih}\delta_{ij}) \leq 1 && (5)\\
\forall i \quad & \textstyle\sum_h y_{ih} \geq \hat{y}_i + Y(1 - \hat{y}_i) && (6)
\end{aligned}$$

Constraint (1) defines the domain of our variables. Constraint (2) states that a service runs on exactly on server. Constraint (3) states that a service can achieve an unscaled yield strictly greater than 0 only on the server on which it runs. Constraint (4) states that a service achieves a yield larger than its minimum required yield. Note that only one of the terms in the summation is non-zero. Constraint (5) states that the fraction of resource j on server h that is allocated to services is at most 1. The expression in the summation is explained as follows. If resource need r_{ij} is fluid, then $\delta_{ij} = 0$ and the fraction of resource j used on server h is $r_{ij} \times y_{ih}$ (the maximum usable fraction multiplied by the yield). If instead resource need r_{ij} is rigid, then $\delta_{ij} = 1$ and the fraction of resource j used on server h is simply r_{ij}. Finally, Constraint (6) states that the minimum yield, Y, is no greater than the yield of any service. This constraint is written so that it is subsumed by Constraint (4) for a service i with $\hat{y}_i = 1$. Recall that such a service must have, by definition, a yield equal to 1 in order for an allocation to be valid. The objective is to *maximize* Y.

3.6. Complexity Analysis

Consider ResAlloc-Dec, the decision problem associated with ResAlloc: Is it possible to find a resource allocation so that the minimum yield is above a given bound, K? Not surprisingly, ResAlloc-Dec is NP-complete. For instance, considering two servers that provide a single resource and services that have rigid needs for that resource, ResAlloc-Dec trivially reduces to 2-Partition, which is known to be NP-complete [20]. We actually obtain a stronger result, by reducing ResAlloc-Dec to 3-Partition, which is NP-complete in the strong sense [20]. This result holds even if all needs are fluid and no service has minimum yield requirements. Due to space limitations, we refer the reader to a technical report for the straightforward proof [57].

4. Algorithms for Solving the Base Problem

4.1. Exact Solution

Solving the MILP formulation of ResAlloc in Section 3.5 provides an exact solution, but can only be done in exponential time. We use a publicly available MILP solver, the Gnu Linear Programming Kit (GLPK), to compute exact solutions for small problem instances (few servers, few services) in under an hour. We denote this exact solution by Opt.

4.2. Greedy Algorithms

In this section we propose greedy algorithms to solve ResAlloc. These algorithms are not identical to greedy algorithms for vector packing [33, 38], due to the presence of fluid resource needs, but inspired by similar ideas. The standard approach is to sort the services in some order, and then to pick a server for each service in order. We consider the following seven options to sort services: (S1) randomly; (S2) sorted by decreasing maximum fluid need; (S3) by decreasing sum of fluid needs; (S4) by decreasing maximum rigid need and constrained fluid need; (S5) by decreasing sum of rigid needs and constrained fluid needs; (S6) by decreasing maximum resource need, whether rigid of fluid; (S7) by decreasing sum of rigid and fluid needs. We do not consider increasing orders since they are known to be inferior to decreasing orders for the vast majority of bin packing problem instances.

We also consider seven options to pick a server for a given service, i, provided that the server can accommodate the service's resource requirements. Let j_f be the index of the resource corresponding to the maximum fluid resource need of service i, i.e., r_{ij_f}. Let j_r be the index of the resource corresponding to the maximum rigid need or constrained fluid need of service i, i.e., r_{ij_r}. Let I_h be the set of the indices of the services already placed on server h. Two options are: (P1) pick server h with the smallest $\max_{i' \in I_h} r_{i'j_f}$; and (P2) pick server h with the smallest $\sum_{i' \in I_h} r_{i'j_f}$. In other words, P1 (resp. P2) places service i on the server that has the smallest maximum (resp. sum) of the fluid needs of services already placed on that server for resource j_f. These two options are oblivious to rigid or constrained fluid resource needs. The other approach is to be oblivious to fluid resource needs. This can be done using standard best fit or worst fit placement, evaluating the load of each server h based on $\max_{i' \in I_h} r_{i'j_r}$ or $\sum_{i' \in I_h} r_{i'j_r}$. We term P3 and P4 the two corresponding best fit approaches, and P5 and P6 the two corresponding worst fit approaches. Finally, one can simply use first fit placement, placing a service on the first server that can accommodate its rigid and constrained fluid needs, which we call P7.

Combining all service sorting and server picking options above, we obtain $7 \times 7 = 49$ different greedy algorithms, which we name GREEDY_SX_PY, where X $\in \{1, 2, 3, 4, 5, 6, 7\}$ and Y $\in \{1, 2, 3, 4, 5, 6, 7\}$. All these algorithms are straightforward to implement with complexity at most $O(N \log N + NH)$ for a fixed number of resource dimensions. These algorithms subsume the greedy algorithms proposed in the applied literature [61, 32, 26].

All these algorithms could be augmented with a backtracking feature (with some bound to avoid exponential complexity) in order to further explore the space of possible solutions when a solution cannot be found. This technique was evaluated in previous work and shown to be unsuccessful for moderately large program instances, even with only two resource dimensions [58]. Thus, we do not attempt backtracking for any of the above algorithms.

4.3. Genetic Algorithm

We implement a genetic algorithm similar to that used in [48] based on the GAlib library [19]. Each chromosome is a 1-D integer array of length N, in which the i-th value is equal to h if service i is allocated to server h. An initial chromosome is obtained by assigning each service to a random server. The mutation operator randomly swaps two services between two different servers. Simply moving a service to a random server, instead of swapping, proved less effective in practice. We use a one-point crossover operator, by which two parent chromosomes are each cut into two segments and two new chromosomes are obtained by concatenating these segments. A new generated chromosome (initial, after mutation, or after crossover) may not correspond to a feasible resource allocation. We allow for infeasible chromosomes in our population. However, after an infeasible chromosome is generated, we use a greedy algorithm that attempts to make the chromosome feasible. This algorithm goes through the servers in an arbitrary order, and for each overloaded server attempts to move services to other less loaded servers. This approach, which reduces the diversity of the chromosome population and biases it toward feasible allocations, proved dramatically beneficial in practice. The fitness of an infeasible genome is defined as the number of servers that are not overloaded in the mapping corresponding to the genome, and is thus between 0 and H. The fitness of a feasible genome is defined as $H(1 + Y)$, where Y is the achieved minimum yield. This fitness is thus between H and $2H$. We use a population size of 100, running for 2,000 generations, with a mutation probability of 0.1 and a crossover probability of 0.25. These parameters were estimated empirically based on calibration experiments with 792 ResAlloc instances (one instance for each experimental scenario, as described in Section 5.1).

4.4. Relaxed LP Solution and its Uses

For large problem instances, the MILP formulation of ResAlloc cannot be solved in reasonable time. For these instances, we instead solve a *relaxed* version of the problem in which all variables are assumed to be rational. The obtained solution is typically infeasible as services could be split across servers. But this solution is useful as its minimum yield is an upper bound on the maximum minimum yield obtained when solving the MILP. It forms a good basis for comparing various heuristics to the optimal: if a heuristic achieves a minimum yield close to the upper bound on the optimal, then it is also close to the optimal. We call this upper bound LPBound. If the rational LP can be solved (i.e., the aggregate resource capacities can meet all rigid and constrained fluid needs), then it has an immediate solution:

$$Y = \min\left(1, \min_{j \in NZ} \frac{H - \sum_i r_{ij}(\hat{y}_i(1-\delta_{ij}) + \delta_{ij})}{\sum_i (1-\hat{y}_i) r_{ij}(1-\delta_{ij})}\right),$$

where NZ is the set of indices $j \in \{1, \ldots, d\}$ such that $\sum_i (1-\hat{y}_i) r_{ij}(1-\delta_{ij})$ is non-zero. This maximum minimum yield is achieved by the trivial allocation $e_{ih} = 1/H$ and $y_{ih} = \frac{1}{H}\left(\hat{y}_i + Y(1-\hat{y}_i)\right)$, for all i and h.

Another use for the solution to the rational LP is that it may point the way toward good solutions of the MILP. A well-known idea is to round off to integer values the rational values assigned to integer variables, in our case the e_{ih} variables. For instance, if the relaxed solution produces some e_{ih} equal to 0.98, it seems likely that this value is equal to 1 in the solution of the MILP. Given a solution to the rational LP, we use the rounding approach in [37]: for each service i, taken in an arbitrary order, allocate service i to server h with probability e_{ih}. For each server h that cannot accommodate service i due to resource constraints, then e_{ih} is set to 0 and other probabilities are scaled accordingly. We call this algorithm RRnd (Randomized Rounding).

The trivial solution given above for the rational LP is a very poor starting point for rounding off e_{ih} values. Since all these values are identical, a service i is equally likely to be allocated to any server, which amounts to a random greedy allocation. A good starting point would be a solution of the rational LP in which e_{ih} values are diverse and distributed over the interval $[0, 1]$. We use GLPK, which uses the simplex algorithm to compute a solution in polynomial time. It turns out that, in practice, this solution leads to e_{ih} values that are well distributed in the interval $[0, 1]$.

One problem with RRnd is that service i may not fit on any server h for which $e_{ih} > 0$ due to resource constraints. In this case the algorithm fails. To remedy this problem we first set each zero e_{ih} to a small value ε (we use $\varepsilon = 0.01$). We call this algorithm RRndNZ (Randomized Rounding Non-Zero). For those problem instances for which RRND provides solutions RRNDNZ should provide nearly identical solutions. But RRNDNZ should also provide solutions for some instances for which RRND fails.

Another approach, termed "diving" in [8], consists in solving the rational LP iteratively, each time fixing the e_{ih} variable closest to 0 or 1 to that value. This requires at most $N \times H$ rational LP resolutions, each time solving a LP with one fewer variable. Another approach, requiring at most N rational LP resolutions, consists in fixing at each iteration all H e_{ih} variables for a given i, picking the i with the largest $\max_h e_{ih}$ at each iteration. We term the first approach SLOWDIVING and the second approach FASTDIVING.

4.5. Vector Packing Algorithms

Resource allocation is a kind of bin packing. Due to multiple resource dimensions, our resource allocation problem is related to the multi-dimensional version of bin packing, or *vector packing*. There is an important difference between RESALLOC and vector packing: our services may have fluid resource needs. This difference can be addressed as follows. Consider an instance of RESALLOC and a fixed value of the yield, Y, that needs to be achieved for each service. Fixing Y amounts to making all resource needs rigid. The problem then becomes exactly vector packing. A binary search on Y can be used to find the highest value for which the vector packing problem can be solved. Given a vector packing algorithm ALG, we term this general approach VP_ALG. We use the following vector packing algorithms, which attempt to place N d-dimensional vectors into (at most) H bins.

Best Fit and First Fit Algorithms [33, 16] – Standard Best Fit (BF) and First Fit (FF) algorithms are among the first algorithms used for solving vector packing problems and both rely on pre-sorting of the input vectors. We use three approaches to sort the vectors, as outlined in [33]: by decreasing sum of the coordinates (SUM), by decreasing maximum of the coordinates (MAX), and by decreasing lexicographical order (LEX). For fixed d, these algorithms can all be implemented straightforwardly with complexity $O(N \log N + NH)$. We obtain 6 new algorithms: VP_BFSUM, VP_BFMAX, VP_BFLEX, VP_FFSUM, VP_FFMAX, and VP_FFLEX. An intriguing heuristic is presented in [38] as an add-on to any algorithm A that first sorts all vectors according to some criteria. Given the assignment of vectors to bins produced by A, one computes a metric called the "degree of dominance", which quantifies, for each dimension, the probability that this dimension causes bin capacities to be exceeded. One then re-sorts all vectors based on a sum of their coordinates weighted by their degrees of dominance, and apply algorithm A with this order. We have implemented this heuristic for all 6 algorithms but did not observe a single case in which it led to an improvement in our experiments over 72,900 problem instances.

Permutation Pack (PP) and Choose Pack (CP) Algorithms [36] – These algorithms attempt to balance the load of the dimensions of each bin. The PP algorithm places each of the N vectors in one of $d!/(d-w)!$ lists, where w is an integer between 1 and d. Each list contains the vectors with a common permutation of their largest w dimensions. For instance, for $w = 2$ and

$d = 3$, there would be 6 lists, for all combinations (i,j), with $i, j \in \{1, \ldots, d\}$. List (i,j) contains the vectors whose i-th coordinate is larger that their j-th coordinate, which is larger than all their other coordinates. Vectors in each list are then sorted according to some criterion. We use four standard options: by decreasing sum of the coordinates (Sum), by decreasing maximum of the coordinates (Max), by decreasing difference of the largest and smallest coordinate (Diff), and by decreasing ratio between the largest and smallest coordinate (Ratio). In [58], the four corresponding increasing orders were evaluated and, unsurprisingly, found to be consistently outperformed by the decreasing orders. The algorithm then starts filling bins with vectors, each time attempting to reduce the resource load imbalance in a bin. This is done by considering the current bin, and determining which w resource dimensions are least loaded for that bin, say, in the case $w = 2$, dimension i and then dimension j. The algorithm then first looks in list (i,j) for the first vector that can fit in the bin, hoping to reduce the resource imbalance. If no such vector can be found, then the algorithm relaxes the ordering of the components and searches in other lists (i.e., trying list (i,k), where k is the third least loaded resource of the bin, etc.). If no vector can fit in the current bin, then a new bin is added and the process is repeated until all vectors are placed in bins. The CP algorithm is a relaxation of the PP algorithm in that it does not enforce any ordering between the w coordinates of vectors, and thus needs "only" $d!/w!(d-w)!$ lists. The empirical results in [36] show that $w = 2$ leads to good results and we use this value in this work. For fixed d and w, both algorithms have complexity $O(N \log N)$. With the CP and PP algorithms, and the four options to sort vector lists, we obtain 8 new algorithms: VP_CPSum, VP_CPMax, VP_CPDiff, VP_CPRatio, VP_PPSum, VP_PPMax, VP_PPDiff, and VP_PPRatio.
The $O(\ln d)$ Guaranteed Algorithm [11] – This polynomial-time algorithm solves a rational linear program formulation of the vector packing problem, which leads to a bounded number of non-integral assignments of vectors to bins. Additional bins are then created in a greedy fashion to accommodate all vectors with non-integral assignments. We do not implement the algorithm in [4], in spite of its impressive $(1 + \ln d)$ guarantee. This algorithm formulates the vector packing problem as a set cover problem. Unfortunately, the instance of the set cover problem can be (and in our case, is) exponential in the size of the instance of the vector packing problem. Although an approximation of the set cover problem instance could be formulated, no guidance is provided in [4]. Furthermore, the algorithm has high complexity regardless. We opt for the simpler guaranteed algorithm in [11] instead, which we call VP_Chekuri.

5. Experimental Results

5.1. Experimental Methodology

We evaluate our algorithms using a collection of randomly generated synthetic problem instances for N services and H servers. We generate instances with d resource dimensions, where d is even. For each service, the first $d/2$ resource dimensions correspond to rigid needs and the last $d/2$ resource dimensions to fluid needs. Our formulation of the resource allocation problem in Section 3 is more general since it allows service i to have a rigid need for resource j ($\delta_{ij} = 1$) and another service $i' \neq i$ to have a fluid need for the same resource ($\delta_{i'j} = 0$). However, in the recent literature, scenarios in which all services express the same type of need for each individual resource are prevalent [32, 9, 70].

All resource needs are sampled from a normal probability distribution with mean μ and standard deviation σ. Each service has a probability ρ to have a QoS requirement. We arbitrarily assume all QoS requirements to be 0.5 (i.e., half the service's fluid needs must be met). Experiments with other values, or with random values for all services, have led to similar conclusions regarding the relative performance of our algorithms. Depending on H and N, the aggregate resource needs of the services may overcome aggregate server capacities in one or more resource dimensions. We parameterize the overall resource load as follows. For each of the $d/2$ rigid resource dimensions, we scale all resource needs in that dimension by a single factor, so that the aggregate server capacities can accommodate the aggregate service needs in that dimension while a given fraction of the aggregate server capacities remains free. We call this free fraction *slack*. A lower value denotes an instance that is more difficult to solve. Some of our generated instances may not have solutions, especially for low slack values and/or large numbers of services with QoS requirements.

Unless specified otherwise, we use $H = 64$, $N = 100, 200, 500$, $d = 2, 4, 6$, $\mu = 0.5$, $\sigma = 0.25, 0.5, 1.0$, $\rho = 0.0, 0.25, 0.5$, and $slack = 0.1, 0.2, \ldots, 0.9$. This corresponds to $1 \times 3 \times 3 \times 1 \times 3 \times 3 \times 9 = 729$ scenarios. For each scenario we generate 100 random samples, for a total of 72,900 individual instances. Algorithm execution times are measured on a dedicated 3.2GHz Intel Xeon processor and averaged over 100 sample problem instances with H and N values as indicated in the text, and with $\mu = 0.5$, $\sigma = 0.5$, $\rho = 0.25$, and $slack = 0.5$.

5.2. Greedy Algorithms

In this section we evaluate the 49 greedy algorithms described in Section 4.2. Algorithm evaluation must account for two criteria: (i) how often an algorithm successfully computes a solution; and (ii) how good that solution is compared to those from other algorithms. For each algorithm we compute two metrics. The first is the *failure rate (fr)*, i.e., the percentage of instances for which it fails to find a solution. The second metric is the *distance from bound (dfb)*, i.e., the difference between the achieved minimum yield and LPBound. The *dfb* is computed for all instances for which the algorithm successfully produces a solution. We report average values over these instances as well as 90th percentile values (i.e., the value below which 90% of the *dfb* values fall). *dfb* values are absolute, and we also present results for relative percent *dfb* values (relative to LPBound). Low values for both metrics are desirable. Over our entire set of 72,900 instances, there are 5,320 instances for which no algorithm was able to compute a valid allocation (7.45% of the instances).

Figure 1 shows one data point for each algorithm, with the x-coordinate being the algorithm's *fr* and the y-coordinate being the algorithm's average relative *dfb*. Algorithms located toward the left and the bottom of the figure are

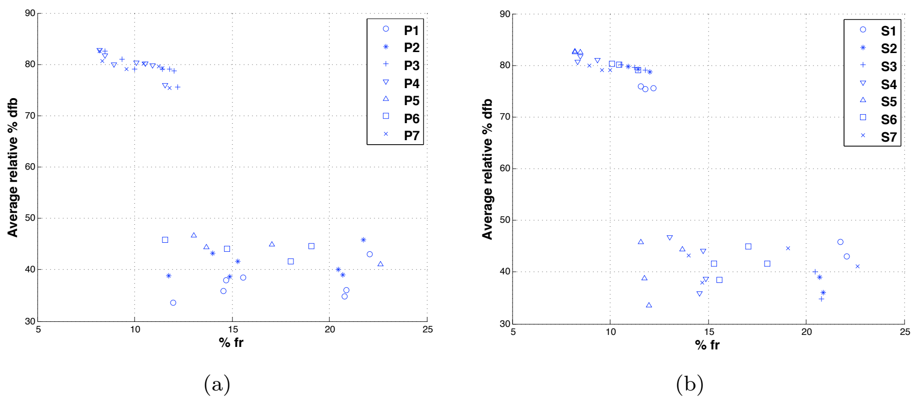

Figure 1: Bi-criteria graphical comparison of all Greedy_Sx_Py algorithms, averaged over all 72,900 instances.

preferable. For better readability, Figure 1(a) differentiates algorithms by their server selection strategy (P1 to P7), while Figure 1(b) differentiates algorithms by their service sorting strategy (S1 to S7).

Algorithms fall in two clusters depending on whether they use the P1, P2, P5, or P6 (Cluster #1), or the P3, P4, or P7 (Cluster #2) server selection strategy. Algorithms in Cluster #1 lead to lower *dfb* (by about 30 points or more), but to higher *fr* (by as much as about 10 points). However, some of the algorithms in Cluster #1 lead to *fr* as low as 12%. The algorithms in Cluster #1 sacrifice *fr* in two ways. Those that use P1 or P2 ignore non-fluid resource needs and thus solely attempt to optimize the yield without attempting to optimize the packing of rigid or constrained fluid needs onto servers. Those that use P5 or P6 do pay more attention to rigid or constrained fluid needs but use a Worst Fit strategy by which they leave resources as free as possible while mapping services to servers. This leads to better opportunities for optimizing the yield, but leads to more failures. In terms of service sorting strategies, S5 and, to a lesser extent, S4 are best. These are the two sorting strategies that consider both rigid needs and constrained fluid needs. Overall, using the maximum of resource needs both for service sorting and for server selection is marginally more effective than using their sum, but no strong empirical claim can be made.

Expectedly, we find that each algorithm, even algorithm Greedy_S1_P7 (random service sorting, random server selection), is best for at least some fraction of our instances. Furthermore, it is difficult to identify clear trends with respect to our instance parameters. We note that the time to execute one of the Greedy_Sx_Py algorithm is relatively low even for moderately large instances, e.g., below 0.1 seconds for instances with $H = 64$ servers and $N = 4,096$ services. A brute-force approach is then to combine all algorithms: run all 49 Greedy_Sx_Py algorithms and pick the best successfully produced resource allocation, if any. We call this approach Greedy.

The Greedy approach may prove too expensive. For instance, for $H =$

Table 1: Average *dfb*, 90th percentile *dfb*, and *fr*, for the Greedy and GA algorithms, over 72,900 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
Greedy	0.16 (30.06%)	0.34 (56.25%)	7.82
GA	0.24 (42.46%)	0.41 (68.18%)	9.57

$1,024$ servers and $N = 16,384$ services, some of the greedy algorithms require up to 3 seconds. Based on our experimental results, we can identify and rank 9 Greedy_Sx_Py algorithms that beat or equal their competitors for more than 15% of the instances (x∈{2, 3, 5, 6, 7} and y=1, and x∈{2, 3, 6} and y=2). We also find that algorithm Greedy_S5_P4 leads to the highest success rate (it fails for only 0.04% of the instances for which some other Greedy_Sx_Py is successful), even though it almost never outperforms the previous 9 algorithms in terms of minimum yield when these algorithms succeed. Therefore, a reasonable approach, which we call GreedyLight, consists in using only these 10 algorithms. Furthermore, GreedyLight tries the 10 algorithms in sequence, stopping as soon as an algorithm produces a resource allocation. The sequence order is that of increasing empirical average *dfb* as observed in our experiments. Out of out the 72,900-5,320 = 67,580 instances for which Greedy succeeds, GreedyLight fails for only 203 instances (or 0.44%). When both algorithms succeed, Greedy outperforms GreedyLight in 20.78% of the cases, in which case if leads to a minimum yield that is relatively better by 20.26% on average. The Greedy and GreedyLight algorithms provide us with good baselines against which to evaluate more sophisticated algorithms.

5.3. Genetic Algorithm

In this section we evaluate the genetic algorithm, GA, described in Section 4.3. Table 1 shows results for the Greedy and GA algorithms, computed over all problem instances. We see that GA is outperformed by Greedy in terms of average *dfb*, 90th percentile *dfb*, and *fr*. Over all feasible instances, GA outperforms Greedy in 9.96% of the cases, and in these cases it leads to a minimum yield on average 26.64% higher than Greedy. By contrast, Greedy outperforms GA in 79.08% of the feasible instances, in which case it leads to a minimum yield that is on average 32.65% higher than GA. We conclude that the genetic algorithm approach is less effective than the greedy approach for our problem. Although we use a population of 100 genomes in GA, we have experimented with population sizes up to 2,000 and did not observe significant improvements. We have seen marginal improvements when increasing the number of generations from 100 to 2,000, suggesting that further increasing the number of generations could be beneficial. However, the execution time of GA is at least one order of magnitude larger than that of Greedy (see Table 3), and increasing the number of generations further is not practical.

Table 2: Average *dfb*, 90th percentile *dfb*, and *fr*, for the LP-based algorithms and Greedy, over 48,600 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
RRnd	0.58 (78.33%)	0.86 (98.52%)	66.56
RRndNZ	0.58 (77.95%)	0.89 (98.28%)	22.02
FastDiving	0.60 (75.03%)	0.84 (94.39%)	78.02
SlowDiving	0.57 (72.75%)	0.81 (93.60%)	77.92
Greedy	0.21 (29.17%)	0.38 (51.49%)	7.50

5.4. LP-based Algorithms

In this section we evaluate the algorithms described in Section 4.4: RRnd, RRndNZ, FastDiving, and SlowDiving. Due to the large execution times of these algorithms we do not present results for $N = 500$, reducing the number of tested instances from 72,900 to 48,600. We compare these algorithms with the Greedy algorithm.

Table 2 shows aggregate results for all five algorithms over all our problem instances. The striking observation is that Greedy largely outperforms all algorithms that rely on solving a rational relaxation of the MILP problem formulation. Greedy fails to compute a solution in only 3,646 of the 48,600 instances, or 7.50%. There is no instance for which Greedy fails and one of its competitors succeeds. By contrast, RRnd, FastDiving, and SlowDiving exhibit high failure rates above 65%. RRndNZ has a much lower failure rate at 22.02%. This is expected since RRndNZ was designed as an improvement to RRnd precisely to reduce the likelihood of failure (in fact, RRndNZ always succeeds when RRnd succeeds). SlowDiving exhibits a slightly lower failure rate than FastDiving, which again is expected as SlowDiving is “more careful” when rounding off rationals to integers.

In terms of *dfb*, we see that Greedy also leads to a drastic improvement relative to the other algorithms, both for the average and the 90th percentile. In spite of their use of more sophisticated methods for rounding rationals to integers, SlowDiving and FastDiving are not significantly closer to LPBound than RRnd and RRndNZ. RRnd and RRndNZ lead to similar *dfb*. SlowDiving provides a marginal improvement over FastDiving.

Greedy runs orders of magnitude faster than the other four algorithms. Table 3 shows average algorithm execution times. Faster execution times could be achieved for all the LP-based algorithm by using a faster (commercial) linear program solver such as CPLEX [15]. Regardless, the execution times would likely still prohibit the use of the algorithms in practice.

The conclusion is that although algorithms that solve a rational relaxation of ResAlloc have been used successfully in the literature, in our context they perform poorly. One intuitive reason for this result is that binary e_{ih} variables are difficult to compute by rounding off rational values. There may simply not be a good way to round off a value of, say, 0.51 or 0.49 to either 0 or 1

Table 3: Average execution times ($H = 64$ and $N = 128, 256, 512$).

	Average Execution Time (sec)		
Algorithm	$N = 128$	$N = 256$	$N = 512$
RRnd	16.30	61.70	255.44
RRndNZ	16.74	61.15	250.83
FastDiving	32.42	113.89	416.32
SlowDiving	382.58	1771.35	6704.79
GA	4.58	7.54	13.79
Greedy	0.05	0.10	0.24

Table 4: Average *dfb*, 90th percentile *dfb*, and *fr*, for the VP-based, Greedy, and GreedyLight algorithms, for 72,900 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
GreedyLight	0.16 (31.49%)	0.35 (56.25%)	8.16
Greedy	0.16 (30.07%)	0.34 (61.01%)	7.73
VP_PPRatio	0.08 (14.54%)	0.17 (28.32%)	15.81
VP_PPDiff	0.08 (13.67%)	0.16 (21.10%)	15.35
VP_FFLex	0.07 (12.85%)	0.15 (27.86%)	15.45
VP_PPMax	0.07 (13.08%)	0.15 (26.67%)	14.99
VP_PPSum	0.07 (12.84%)	0.15 (26.39%)	14.93
VP_CPRatio	0.07 (11.09%)	0.14 (21.21%)	11.45
VP_BFLex	0.06 (12.15%)	0.14 (27.10%)	13.75
VP_CPDiff	0.06 (10.19%)	0.12 (21.10%)	8.70
VP_CPMax	0.05 (10.10%)	0.11 (20.60%)	8.43
VP_CPSum	0.05 (9.92%)	0.11 (20.40%)	8.20
VP_BFMax	0.04 (11.39%)	0.11 (29.40%)	8.48
VP_FFMax	0.04 (11.26%)	0.11 (29.33%)	8.33
VP_BFSum	0.04 (10.95%)	0.10 (28.72%)	7.91
VP_FFSum	0.04 (10.95%)	0.10 (28.40%)	7.91

without leading to a schedule that is far from the optimal schedule. Greedy successfully solves more instances, leads to better minimum yields, and runs orders of magnitude faster. In the rest of this paper we exclude results for RRnd, RRndNZ, FastDiving, and SlowDiving.

5.5. Vector Packing Algorithms

In this section we evaluate the algorithms in Section 4.5, which use a vector packing (VP) approach. We also include Greedy and GreedyLight in this comparison. We do not present results for VP_Chekuri. Due to its computational cost, we ran this algorithm only on instances with $N = 100$. VP_Chekuri leads to much higher failure rates than all its competitors and lower yields in all instances. It is also orders of magnitude more computationally expensive in practice, due to solving a large LP at each iteration of the binary search.

Table 4 summarizes the results averaged over all 72,900 instances. Rows of the table are sorted by decreasing average *dfb* and, for equal *dfb*, by decreasing *fr*. In terms of *dfb*, GREEDY and GREEDYLIGHT are outperformed by all VP-based algorithms. Furthermore, both have failure rates that are not significantly better than those of the least failure-prone VP-based algorithms. All VP-based algorithms exhibit comparable behavior, with no clear clustering of the algorithms when looking at averages. We seek more insight into our results using one-to-one algorithm comparisons via a *domination* relationship. For two algorithms A and B, we define the following two measures: (i) $\mathcal{S}_{A,B}$: the percentage of instances for which A succeeds and B fails, which takes a positive value; and (ii) $\mathcal{Y}_{A,B}$: the average percent minimum yield difference between A and B, relative to the minimum yield achieved by B, computed on instances for which both algorithms succeed, which can take a positive or negative value. For both measures, a positive value means an advantage of A over B. We say that, based on our experiments, "algorithm A dominates algorithm B" if $\mathcal{S}_{A,B} \geq 0.5\%$ and $\mathcal{Y}_{B,A} \leq 3\%$, or, $\mathcal{S}_{B,A} \leq 0.5\%$ and $\mathcal{Y}_{A,B} \geq 3\%$. In other words, algorithm A dominates algorithm B if it is significantly more successful and not significantly less effective at maximizing minimum yield, or if it is significantly more effective at maximizing minimum yield and not significantly less successful. We say that "algorithms A and B are equivalent" if neither algorithm dominates the other. We picked 0.5% to mean "not significant" and 3% to mean "significant." We experimented with values higher that 3% for significance. We found that for these values very few dominance relationships could be established, due to many of the algorithms being close to each other in terms of *dfb* and *fr*.

We established domination relationships based on our experimental results considering all our experiments ($d \in \{2, 4, 6\}$), the two subsets for $d \in \{2, 4\}$ and $d \in \{4, 6\}$, and the three subsets for $d = 2$, $d = 4$, and $d = 6$, for a total of $1 + 2 + 3 = 6$ result subsets. The goal of considering these subsets is to determine whether the number of resource dimensions has an impact on the relative performance and the algorithms.

We found that each Permutation Pack (PP) algorithm is dominated by its Choose Pack (CP) counterpart, for all result subsets. We found that VP_CPRATIO is dominated by VP_CPDIFF across all result subsets. Among the remaining three CP algorithms, VP_CPSUM is the only one that is not dominated for any result subset. VP_CPDIFF is not dominated for subsets $d = 2, 4, 6$, $d = 2, 4$, and $d = 2$, while VP_CPMAX is not dominated for subsets $d = 4, 6$, $d = 4$, and $d = 6$. This indicates that VP_CPMAX is preferable to VP_CPDIFF for problems with more resource dimensions, while the situation is reversed for problems with lower resource dimensions. Regardless, we conclude that VP_CPSUM is the algorithm of choice among all PP and CP algorithms.

Among the algorithms that use a Best Fit or First Fit approach, we found that algorithms using lexicographical ordering (VP_FFLEX and VP_BFLEX) are each dominated by both of their counterparts on all result subsets. VP_FFSUM and VP_FFMAX are equivalent for high resource dimensions (result subsets $d = 4, 6$ and $d = 4$), but VP_FFSUM dominates VP_FFMAX for all other result subsets. Conclusions are identical for VP_BFSUM and VP_BFMAX. The

Table 5: Some of the $\mathcal{S}$ and $\mathcal{Y}$ values pertaining to the VP_CPSUM algorithm, computed over all our problem instances.

Measure	Value (%)
$\mathcal{S}_{\text{VP\_CPSUM,VP\_FFSUM}}$	0.02
$\mathcal{Y}_{\text{VP\_CPSUM,VP\_FFSUM}}$	6.93
$\mathcal{S}_{\text{VP\_FFSUM,VP\_CPSUM}}$	0.56
$\mathcal{Y}_{\text{VP\_FFSUM,VP\_CPSUM}}$	-1.11
$\mathcal{S}_{\text{VP\_CPSUM,VP\_BFSUM}}$	0.04
$\mathcal{Y}_{\text{VP\_CPSUM,VP\_BFSUM}}$	7.28
$\mathcal{S}_{\text{VP\_BFSUM,VP\_CPSUM}}$	0.67
$\mathcal{Y}_{\text{VP\_BFSUM,VP\_CPSUM}}$	-1.29

Measure	Value (%)
$\mathcal{S}_{\text{VP\_CPSUM,GREEDY}}$	0.01
$\mathcal{Y}_{\text{VP\_CPSUM,GREEDY}}$	42.23
$\mathcal{S}_{\text{GREEDY,VP\_CPSUM}}$	0.99
$\mathcal{Y}_{\text{GREEDY,VP\_CPSUM}}$	-22.34
$\mathcal{S}_{\text{VP\_CPSUM,GREEDYLIGHT}}$	0.00
$\mathcal{Y}_{\text{VP\_CPSUM,GREEDYLIGHT}}$	68.61
$\mathcal{S}_{\text{GREEDYLIGHT,VP\_CPSUM}}$	0.66
$\mathcal{Y}_{\text{GREEDYLIGHT,VP\_CPSUM}}$	-24.05

VP_FFSUM and VP_BFSUM algorithms are equivalent on all result subsets.

We are left with the VP_CPSUM, VP_BFSUM, and VP_FFSUM algorithms. These three algorithms are equivalent on all our result datasets, by our definition of equivalence. The left-hand side of Table 5 shows the $\mathcal{S}$ and $\mathcal{Y}$ measures for VP_CPSUM vs. VP_FFSUM and VP_BFSUM. We see that the $\mathcal{S}$ values 0.56% and 0.67% are only slightly above our 0.5% insignificance threshold. When considering all our problem instances, it turns out that VP_CPSUM is outperformed by VP_FFSUM (resp. VP_BFSUM) for 68.97% (resp. 76.26%) of instances for which both algorithms succeed. However, in these cases, its minimum yield is only outperformed by 4.01% (resp. 3.93%) on average. However, when VP_CPSUM outperforms VP_FFSUM (resp. VP_BFSUM) it does so by 27.00% (resp. 18.70%) on average. We conclude that the best algorithm among the VP-based ones is VP_CPSUM. In terms of computational demands, the execution times of the three algorithms are comparable. For instances with $N \times H = 8,388,608$ (e.g., $N = 8,192$ services running on a cluster with $H = 1024$ servers), the average execution time of VP_CPSUM, VP_FFSUM, and VP_BFSUM are 1.38, 1.50, and 1.62 seconds, respectively.

VP_CPSUM is also preferable to the GREEDY and GREEDYLIGHT approaches, as can be seen in the right-hand side of Table 5. While $\mathcal{S}_{\text{GREEDY,VP\_CPSUM}}$ and $\mathcal{S}_{\text{GREEDYLIGHT,VP\_CPSUM}}$ are low (above the 0.5% threshold but below 1%), $\mathcal{Y}$ values show that VP_CPSUM largely outperforms GREEDY and GREEDYLIGHT in terms of minimum yield. Furthermore, VP_CPSUM is also less computationally demanding than GREEDY and GREEDYLIGHT. For the aforementioned instances with $N \times H = 8,388,608$, the execution time of VP_CPSUM is 1.38 seconds while that of GREEDY is 54.56 seconds and that of GREEDYLIGHT is 1.75 seconds.

We conclude that among all the algorithms considered in this work, the VP_CPSUM algorithm is the algorithm that should be used in practice for computing resource allocations. This conclusion holds when taking further subsets of our results with respect to the σ, ρ, and *slack* parameters.

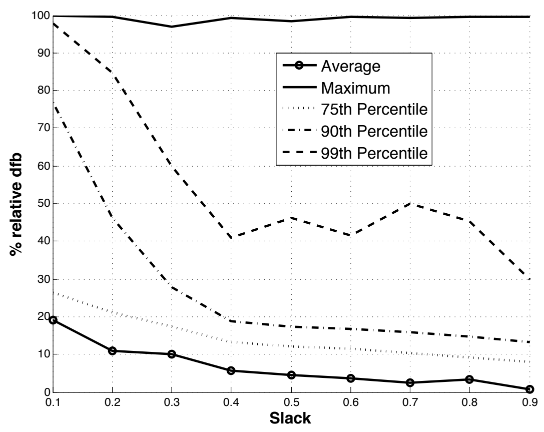

Figure 2: Percent relative *dfb* values for VP_CPSum vs. the *slack*.

Table 6: VP_CPSum's *dfb* values and relative *dfb* values (in parentheses), when fixing the memory slack and one of the parameters defining the instances.

Fixed param.	Param. Value	*slack* = 0.1				*slack* = 0.4			
		average		90th perc.		average		90th perc.	
d	2	0.05	(16.51%)	0.13	(74.82%)	0.00	(0.19%)	0.24	(10.7%)
	4	0.09	(16.20%)	0.22	(41.61%)	0.03	(4.00%)	0.08	(15.61%)
	6	0.18	(31.83%)	0.57	(94.93%)	0.09	(13.15%)	0.17	(22.19%)
ρ	0.00	0.10	(16.90%)	0.49	(79.83%)	0.03	(5.14%)	0.10	(13.6%)
	0.25	0.08	(25.45%)	0.17	(79.13%)	0.04	(7.00%)	0.11	(18.92%)
	0.50	0.09	(14.91%)	0.22	(54.87%)	0.06	(1.13%)	0.16	(24.25%)
N	100	0.10	(10.29%)	0.21	(21.29%)	0.06	(6.00%)	0.17	(17.2%)
	200	0.15	(26.60%)	0.51	(84.90%)	0.05	(9.78%)	0.10	(20.12%)
	500	0.02	(15.02%)	0.11	(77.77%)	0.00	(0.01%)	0.02	(20.54%)
σ	0.25	0.10	(20.83%)	0.33	(80.62%)	0.03	(4.38%)	0.11	(17.30%)
	0.50	0.09	(20.09%)	0.20	(76.19%)	0.04	(6.36%)	0.13	(18.79%)
	1.00	0.08	(16.51%)	0.21	(70.71%)	0.04	(6.03%)	0.14	(19.69%)

5.6. Impact of Instance Parameters

In this section we study the effects of our instance parameters on the behavior of VP_CPSUM. Figure 2 plots average, maximum, 75th percentile, 90th percentile, and 99th percentile of percent relative *dfb* values for VP_CPSUM versus the *slack*. We see that *dfb* average values roughly decrease as the *slack* increases. This is expected since higher *slack* means an easier resource allocation problem. In our most difficult instances, $slack = 0.1$, VP_CPSUM's relative *dfb* is on average 19.23%, which is reasonably close to LPBOUND. The 75th percentile curve is close to the average curve, denoting that for the bulk of our experiments VP_CPSUM is still close to LPBOUND. The 90th and 99th percentile curves are expectedly higher, and the maximum curve remains close to 1.0. This means that regardless of the *slack* value, i.e., of the difficulty of the problem, there are still some instances that are hard to solve for our algorithm.

Table 6 shows VP_CPSUM's absolute and relative *dfb* values for subset of the instances when fixing one of the parameters that define our instances. The table shows average and 90th percentile values, for $slack = 0.1$ and $slack = 0.4$. We see that higher d increases the *dfb*. In the difficult case $slack = 0.1$, the relative *dfb* is at most 31.83% when $d = 6$. Higher values of ρ do not have much of an effect on the already difficult $slack = 0.1$ case, but do make the easier $slack = 0.4$ case more difficult. This is because with more services having QoS requirements, the resource allocation problem becomes more difficult. The σ parameter does not have a significant impact on the results. High values of the N parameter lead to low *dfb* values. This is because with more services, and keeping the *slack* constant, the resource allocation problem becomes easier (many smaller jobs are easier to pack into servers than fewer bigger jobs). Overall, this table demonstrates that, even when considering several subsets of our results, VP_CPSUM is not far from LPBOUND on average (at most 31.83%).

An important observation here is that VP_CPSUM is no further from OPT than from LPBOUND since LPBOUND is an upper bound on OPT. We conclude that VP_CPSUM produce resource allocations that are, on average, within roughly 30% of the optimal.

5.7. OPT *vs.* LPBOUND

So far we have used *dfb* as our measure of goodness for resource allocations, that is the distance to the LPBOUND upper bound on the optimal minimum yield. While a low *dfb* is certainly desirable, there remains the question of how tight the upper bound is. In this section we compare VP_CPSUM and LPBOUND to the optimal solution OPT computed by solving the MILP formulation of the resource allocation problem given in Section 3.5. Since solving a MILP takes exponential time, we use a set of "small" instances with $H = 4$, $N = 8, 10, 12$, and setting the other parameter values as previously. We found that out of these 72,900 new instances, the MILP solver in GLPK failed to find a solution for 4,325 instances, or 5.93%. We assume that these instances have no feasible solution; more than 80% of them have $slack = 0.1$.

Figure 3 plots relative percent differences between VP_CPSUM and OPT, and between OPT and LPBOUND, both for the average and the 90th percentile.

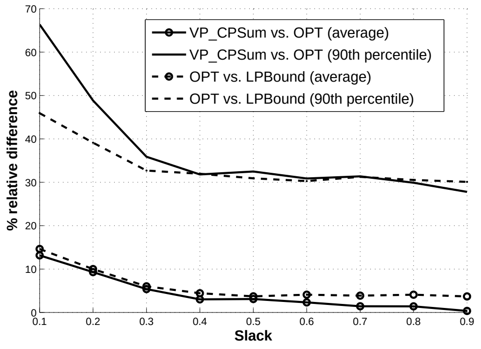

Figure 3: Percent relative *dfb* values for VP_CPSum and Opt, vs. *slack*.

We see that, expectedly, all values decrease as *slack* increases. Let us first examine the difference between Opt and LPBound (dashed lines on the figure). For the most difficult scenarios, i.e., $slack = 0.1$, Opt is on average 14.63% away from LPBound. The 90th percentile for $slack = 0.1$ is reasonable at 45.91%, showing that for the bulk of the instances LPBound is a relatively tight upper bound. We conclude that, at least for small instances, LPBound is a tight upper bound on optimal. Turning our attention to the distance between VP_CPSum and Opt (solid lines), we see that roughly the same observations can be made even though the 90th percentile values for low *slack* are a bit higher (up to 66.35% for $slack = 0.1$). We conclude that VP_CPSum is roughly as far from Opt as Opt is from LPBound. In other terms, the relative difference between VP_CPSum and Opt is about half of that between VP_CPSum and LPBound. While impossible to verify, if this observation also holds true for large problem instance, then halving the values in Figure 2 and Table 6 would provide reasonable estimates of how far VP_CPSum is from optimal.

5.8. Optimizing Average Yield

Once an allocation with a given maximum minimum yield, say $\mathcal{Y}$, has been produced, there may be excess resources available. To further improve resource utilization one can then maximize the average yield while preserving $\mathcal{Y}$ as the maximum minimum yield. This optimization can be framed as a MILP, simply replacing the objective function by the average yield, and adding the constraint $Y \geq \mathcal{Y}$. Unfortunately, the MILP cannot be solved in polynomial time and solving it would require developing and evaluating a number of heuristics. Instead, we opt for a simple solution: we enforce that the e_{ih} values computed by the minimum yield maximization procedure be kept constant. In other words, we do not allow average yield maximization to change the mapping of services to servers. We only allow it to change allocated resource fractions. In this case, the MILP becomes a rational LP since all integer variables have become constants. It turns out that a simple greedy algorithm solves this LP. First, set the

Table 7: Average cluster utilization.

		Util. (%)	Opt. Util. (%)
$d = 2$	$N = 100$	77.75	78.32
	$N = 200$	98.03	99.98
	$N = 500$	95.70	99.97
$d = 4$	$N = 100$	72.87	77.06
	$N = 200$	88.60	94.51
	$N = 500$	94.76	96.74
$d = 6$	$N = 100$	70.07	75.77
	$N = 200$	83.15	90.92
	$N = 500$	93.36	95.82

yield of each service to the minimum yield and update their resource fractions accordingly. Then, for each server, evaluate which service could get the highest yield increase given the remaining available resource fractions. Increase the yield of that service as much as possible. If free resources remain, repeat that procedure until no service can see its yield further increased. The optimality of this process is easily proved via a typical exchange argument.

Note that if the system is truly undersubscribed, then some servers can be turned off. Our algorithm can be used to determine resource allocation with different possible numbers of servers and thus provide guidance on whether and how many servers could be turned off without impacting the minimum yield in an unacceptable manner.

For each resource allocation we compute an overall cluster *utilization* metric. This metric accounts for the aggregate percentage of resources used in the cluster, excluding resources that correspond exclusively to rigid needs. Indeed, the utilization of rigid resources is dictated by the rigid needs of the services, since in a successful allocation all rigid needs must be met. This utilization is then a fixed quantity for a given problem instance, regardless of the resource allocation algorithm used, and we simply do not account for it.

Table 7 shows average utilization values computed over several subsets of our 72,900 instances, before and after average yield maximization, when using the VP_CPSum algorithm. We see that as the number of resource dimensions increases, cluster utilization decreases. This is expected as with more resource dimensions vector packing is more difficult. We also see that, with the exception of the $d = 2$ results for $N = 200$ and $N = 500$, a larger number of services increases utilization. Again, this is expected given that it is easier to pack many small vectors than fewer larger vectors. Results regarding how utilization varies with slack (not shown in the table) show that utilization improves marginally as the slack increases. We found that instance parameters ρ and σ had negligible impact on cluster utilization. Finally, we see that our average yield optimization step does improve cluster utilization noticeably. Larger improvements could be achieved by removing the constraint that the mapping of services to servers be unchanged, but this would require the development of efficient average yield optimization heuristics.

6. Multi-VM Services

So far we have assumed that each service consists of a single VM instance. There are, however, at least two compelling reasons why multi-VM services may be useful. First, a service could be implemented as a data-parallel application that consists of identical communicating tasks, with each task running inside a VM instance. This may be necessary in case the service requires an aggregate amount of memory beyond what a single server can provide, which is often the case for data-parallel applications in scientific domains. Second, a service may naturally consist of two or more components that must live within different VM instances and that have related resource needs (e.g., in steady-state a component uses half as much resources as another component). We discuss both these cases hereafter, explaining how our approach can be applied to each.

6.1. Data-parallel Services

Our algorithms, and thus the VP_CPSum algorithm, can be used almost directly to handle the data-parallel service scenario. In such a scenario, the service computes as fast as its slowest task. As a result, all tasks, which have identical resource needs, should be given identical resource fractions to avoid wasting resources. All tasks within a service should then experience the same yield. All tasks in the service can then be considered as individual services. This approach amounts to equating the yield of a data-parallel service with that of each of its individual tasks. Therefore, there is no incentive to implement a service in a data-parallel fashion simply for the purpose of achieving higher yield. Furthermore, our algorithms for single-task services can be employed *directly* (see for instance the experimental results in [58]).

There is a single difference between our approach for single-task services and that for data-parallel services. Recall that after computing a resource allocation that maximizes the minimum yield, our approach proceeds with an average yield maximization step. In Section 5.8, we have seen that, in the case of single-task services, average yield optimization can be formulated as a MILP. When not allowing average yield maximization to change the mapping of services to servers (but only the resource fractions), this MILP becomes a rational LP. In the case of single-VM services, it turns out that the rational LP can be solved directly via a simple greedy algorithm. Unfortunately, the same algorithm cannot be applied to data-parallel services. One must then solve the rational LP, which can be done in polynomial time. For the sake of completeness we now give the formulation of this LP.

Let T_i be the number of tasks of service i. Let us consider a given mapping of tasks to servers with an achieved minimum yield Y. The mapping of a task to a server is fully defined via a binary value e_{ith} that is 1 if task t, $0 \leq t < T_i$, of service i is allocated to server h, and 0 otherwise. Defining y_{ith} as the unscaled yield of task t of service i on server h, we can now write the following constraints, which are similar to the ones in Section 3.5. The main difference is that the e_{ith} values and the Y value are constants rather than variables:

$$\forall i,t,h \qquad y_{ith} \in \mathbb{Q} \tag{7}$$
$$\forall i,t,h \qquad 0 \leq y_{ith} \leq e_{ith} \tag{8}$$
$$\forall i,t \qquad \textstyle\sum_h y_{ith} \geq \hat{y}_i \tag{9}$$
$$\forall i,t>0 \qquad \textstyle\sum_h y_{ith} = \sum_h y_{i0h} \tag{10}$$
$$\forall h,j \qquad \textstyle\sum_{i,t} r_{ij}(y_{ith}(1-\delta_{ij}) + e_{ith}\delta_{ij}) \leq 1 \tag{11}$$
$$\forall i,t \qquad \textstyle\sum_h y_{ith} \geq \hat{y}_i + Y(1-\hat{y}_i) \tag{12}$$
$$X = \sum_{i \ s.t. \ \hat{y}_i<1,t} \frac{(\sum_h y_{ith}) - \hat{y}_i}{T_i(1-\hat{y}_i)} \tag{13}$$

Constraint (10) ensures that all tasks within a job have the same unscaled yield, which implies identical resource allocations. Constraint (12) ensures that the minimum yield is not reduced by average yield optimization. Constraint (13) defines X as the sum of the yields of all services with a minimum yield strictly lower than 1 (all other services have by definition a scaled yield equal to 1). Therefore X is linearly and positively correlated with the average yield computed over all services. The optimization objective is to maximize X. All variables, the y_{ith} values and X, are rational making this program solvable in polynomial time.

6.2. Multi-Instance Services

In the case of a service that is implemented with multiple components each running inside its own VM instance, our approach can be easily used if a linear relationship exists between the fluid resource needs of service components. For instance, consider a simple scenario in which servers provide two resources, $r1$ and $r2$. Consider two components of the same service, A and B, each of them in its own VM instance. Say that component A's fluid need in resource $r1$ is 40% and component A's fluid need in resource $r2$ is 20%. If, for each resource, component B requires half the amount required by component A then one can just set its fluid needs to 20% and 10% for resources $r1$ and $r2$, respectively, and treat both components as independent services. Before the average yield optimization step, all services have the same yield. Similarly to the data-parallel service case, the only modification to our approach is the resolution of a rational LP for average yield optimization. The fact that one component requires half as many resources as that required by another component could be specified by an operator. Alternately, the operator could simply specify that the two components are related, and the relationship could be discovered using the techniques described in Section 3.3.

As recognized in [60], multi-tier services raise a number of challenges. First, the relationship between the resource consumption of tiers are not necessarily linear or even uniform across resource dimensions (e.g., component A requires half the CPU of component B but twice the bandwidth), in which case our approach would need to be modified. Second, in a multi-tier scenario the number of instances for each tier may not be specified a priori. Automatically deciding the number of instances is outside the scope of this paper, and raises the difficult "shifting bottleneck" issue identified in [60].

7. Conclusion

In this paper we have studied the resource allocation problem in shared hosting platforms for static workloads with servers that provide multiple types of resources. We have given a formulation of the problem that supports a mix of QoS and best-effort scenarios, and that attempts to maximize a generic objective function, the minimum yield. We have explained how an (in practice reasonably tight) upper bound on the optimal yield can be computed, and how the average yield can be maximized as a way to increase cluster utilization. We have proposed and evaluated several classes of algorithms over a wide range of simulation scenarios. We have found that performing a binary search over the yield and solving the resource allocation problem for a fixed yield using a vector packing algorithm is the best approach. Vector packing algorithms that reason on the sum of the resource needs of the services are the most effective. Among these algorithms the Chose Pack vector packing algorithm from [36] runs quickly and is the most effective. Most notably, it outperforms a greedy approach that combines many greedy algorithms, as well as linear program relaxations and a genetic algorithm approach. In conclusion, we have found an algorithm that runs in only a few seconds and that computes resource allocations that are close to the optimum.

The scope of this paper is static workloads, i.e., workloads for which the number of instances per service and the resource needs of services do not change throughout time. These assumptions obviously do not hold in practice and resource allocations for most services need to be adapted throughout time as demands for the services increases or decreases. To this end, our algorithm can be used to recompute appropriate resource allocations periodically or based on particular events. A well-known problem with this approach is that the newly computed allocation could be widely different from the previous one, possibly leading to unnecessary "shuffling" of VM instances across servers [61]. There is therefore a need for a way to mitigate the overhead of resource allocation adaptation. One option is to compute resource allocations that are likely to delay the need for allocation adaptation as much as possible [2]. Another, complementary, option is to adapt the resource allocation while attempting to minimize the amount of change [22, 32, 9]. The development of such techniques for our resource allocation problem and its novel components (i.e., minimum yield maximization, arbitrary multiple resource dimensions, mix of QoS and best-effort scenarios) is a natural extension of this work.

Acknowledgments

The authors thank the reviewers for their comments, which led to a much improved article. A preliminary version of portions of this material appeared in proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 09) [58].

References

[1] U.S. Environmental Protection Agency. Report to Congress on Server and Data Center Energy Efficiency. `http://repositories.cdlib.org/lbnl/LBNL-363E/`, August 2007.

[2] S. Ali, J.-K. Kim, H. J. Siegel, and A. A. Maciejewski. Static heuristics for robust resource allocation of continuously executing applications. *J. of Parallel and Distributed Computing*, 68:1070–1080, 2008.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proc. of the ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[4] N. Bansal, A. Caprara, and M. Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *Foundations of Computer Science*, pages 697–708, 2006.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 164–177, October 2003.

[6] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *Proc. of the Symp. On Discrete Algorithms*, pages 270–279, 1998.

[7] M. Bichler, T. Setzer, and B. Speitkamp. Capacity Planning for Virtualized Servers. In *Proc. of the 16th Annual Workshop on Information Technologies & Systems (WITS)*, 2006.

[8] A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.

[9] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. Utility-based placement of dynamic web applications with fairness goals. In *IEEE Network Operations and Management Symposium*, pages 9–16, 2008.

[10] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, 2001.

[11] C. Chekuri and S. Khanna. On multi-dimensional packing problems. *SIAM Journal on Computing*, 33(4):837–851, 2004.

[12] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Translating Service Level Objectives to lower level policies for multi-tier services. *Cluster Computing*, 11(3):299–311, 2008.

[13] L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proc. of the Usenix Annual Technical Conference*, 2005.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. of the Symp. on Networked Systems Design and Implementation*, pages 273–286, 2005.

[15] ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[16] J. Csirik, J. B. G. Frenk, M. Labbe, and S. Zhang. On multidimensional vector bin packing. *Acta Cybernetica*, 9(4):361–369, 1990.

[17] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[18] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[19] GAlib: A C++ Library of Genetic Algorithm Components. `http://lancet.mit.edu/ga/`, 2010.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[21] D. Gmach. *Managing Shared Resource Pools for Enterprise Applications.* PhD thesis, Technische Universität München, Fakultät fr̈ Informatik, 2009.

[22] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *Proc. of the IEEE Intnl. Conf. on Dependable Systems and Networks*, pages 326–335, June 2008.

[23] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *Proc of the 10th IEEE Intnl. Symp. on Workload Characterization*, pages 171–180, Sept. 2007.

[24] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Resource pool management: Reactice versus proactive or let's be friend. *Computer Networks*, 53:2905–2922, 2009.

[25] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. Chase, and J. Albrecht. Harnessing Virtual Machine Resource Control for Job Management. In *Proc. of the 1st Workshop on System-level Virtualization for High Performance Computing*, Nov. 2007.

[26] J. Gueyoung, K. Joshi, and M. Hiltunen. Performance Aware Regeneration in Virtualized Multitier Applications. In *Proc. of Proactive Failure Avoidance Recovery and Maintenance (PFARM)*, June 2009.

[27] D. Gupta, L. Cherkasova, and A. Vahdat. Comparison of the Three CPU Schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 35(2):42–51, September 2007.

[28] D. Gupta, R. Gardner, and L. Cherkasova. XenMon: QoS Monitoring and Performance Profiling Tool. Technical Report HPL-2005-187, Hewlett-Packard Labs, 2005.

[29] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum. Sharing Networked Resources with Brokered Leases. In *Proc. of the USENIX Technical Conference*, June 2006.

[30] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proc. of the USENIX Annual Technical Conf.*, June 2006.

[31] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, October 2006.

[32] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic Placement for Clustered Web Applications. In *Proceedings of the 15th International Conference on the World Wide Web*, pages 595–604, 2006.

[33] L. T. Kou and G. Markowsky. Multidimensional Bin Packing Algorithms. *IBM Journal of Research and Development*, 21(5):443–448, 1977.

[34] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang. Power and Performance Management of Virtualized Computing Environments via Lookahead Control. *Cluster Computing*, 12(1):1–15, 2009.

[35] A. Legrand, A. Su, and F. Vivien. Minimizing the Stretch when Scheduling Flows of Divisible Requests. *J. of Scheduling*, 11(5):381–404, 2008.

[36] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proc. of the Intl. Conf. on Parallel Processing*, pages 404–412, 1999.

[37] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. Steady-State Scheduling of Multiple Divisible Load Applications on Wide-Area Distributed Computing Platforms. *Intl. J. of High Performance Computing Applications*, 20(3):365–381, 2006.

[38] K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2):131–149, 1977.

[39] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proc. of the 21st Large Installation System Administration Conference*, pages 167–181, Nov. 2007.

[40] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 265–286, 2008.

[41] Microsoft Virtual PC. `http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx`.

[42] K. Nesbit, J. Laudon, and J. Smith. Virtual Private Caches. In *Proc. of the Symp. on Computer Architecture*, 2007.

[43] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Proc. of Cloud Computing and Its Applications*, Oct. 2008.

[44] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in Virtual Machine Monitors. In *Proc. of the ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environment (VEE)*, March 2008.

[45] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef. Performance management for cluster-based web services. *J. on Selected Areas in Communications*, 23(12):2333–2343, Dec. 2005.

[46] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 289–302, 2007.

[47] L. Ramakrishnan, L. Grit, A. Iamnitchi, D. Irwin, A. Yumerefendi, and J. Chase. Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control. In *Proc. of the International Conference for High Performance Computing Networking, Storage, and Analysis*, 2006.

[48] J. Rolia, A. Andrzejak, and M. F. Arlitt. Automating Enterprise Application Placement in Resource Utilities. In *Proc. of the 4th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, 2003.

[49] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak. A capacity management service for resource pools. In *Proc. of the 5th international Workshop on Software and Performance*, pages 229–237, 2005.

[50] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt. Toward Effective Multi-capacity Resource Allocation in Distributed Real-time and Embedded Systems. In *Proceedings of the 11th Symposium on Object Oriented Real-Time and Distributed Computing*, 2008.

[51] P. Ruth, R. Junghwan, D. Xu, R. Kennell, and S. Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *Proc. of the IEEE Intl. Conf. on Autonomic Computing*, pages 5–14, June 2006.

[52] D. Schanzenbach and H. Casanova. Accuracy and Responsiveness of CPU Sharing Using Xen's Cap Values. Technical Report ICS2008-05-01, Computer and Information Sciences Dept., University of Hawai'i at Mānoa, May 2008. available at `http://www.ics.hawaii.edu/research/tech-reports/ICS2008-05-01.pdf/view`.

[53] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[54] P. Shivam, S. Babu, and J. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[55] Y. Song, H. Wang, Y. Li, B. Feng, and Y. Sun. Multi-Tiered On-Demand Resource Scheduling for VM-Based Data Center. In *Pro. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 148–155, 2009.

[56] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proc. of the 2nd Symposium on Networked Systems Design & Implementation*, pages 71–84, 2005.

[57] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource Allocation Using Virtual Clusters. Technical Report ICS2008-09-01, Information and Computer Sciences Dept., University of Hawai'i at Mānoa, Sept. 2008.

[58] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource Allocation using Virtual Clusters. In *Proc. of the 9th IEEE Symposium on Cluster Computing and the Grid (CCGrid'09)*, May 2009.

[59] M. Stillwell, F. Vivien, and H. Casanova. Dynamic Fractional Resource Scheduling for HPC Workloads. In *Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2010.

[60] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, 2008.

[61] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239–254, 2002.

[62] Virtualcenter. `http://www.vmware.com/products/vi/vc`, 2008.

[63] VMware. `http://www.vmware.com/`.

[64] R. Wang and N. Kandasamy. A distributed control framework for performance management of virtualized computing environments: some preliminary results. In *Proc. of the 1st workshop on Automated Control for Datacenters and Clouds*, pages 7–12, 2009.

[65] A. Warfield, S. Hand, T. Harris, and I. Pratt. Isolation of Shared Network Resources in XenoServers. Technical Report PDN-02-2006, PlanetLab Project, November 2002.

[66] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture*, February 2007.

[67] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of the 9th International ACM/IFIP/USENIX Middleware Conference*, 2008.

[68] *Proc. of the 1st USENIX Workshop on I/O Virtualization (WIOV)*, Dec. 2008.

[69] Citric XenServer Enterprise. `http://www.xensource.com/products/Pages/XenEnterprise.aspx`, 2008.

[70] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. In *Proceedings of the International Conference on Autonomic Computing (ICAC'08)*, pages 172–181, June 2008.