RTP: Robust Tenant Placement for Elastic In-Memory Database Clusters

Jan Schaffner*, Tim Januschowski‡, Megan Kercher‡, Tim Kraska†,
Hasso Plattner*, Michael J. Franklin§, Dean Jacobs‡

*Hasso Plattner Institute, Potsdam, Germany
‡SAP AG Walldorf, Germany
†Brown University
§AMPLab, UC Berkeley

ABSTRACT

In the cloud services industry, a key issue for cloud operators is to minimize operational costs. In this paper, we consider algorithms that elastically contract and expand a cluster of in-memory databases depending on tenants' behavior over time while maintaining response time guarantees.

We evaluate our tenant placement algorithms using traces obtained from one of SAP's production on-demand applications. Our experiments reveal that our approach lowers operating costs for the database cluster of this application by a factor of 2.2 to 10, measured in Amazon EC2 hourly rates, in comparison to the state of the art. In addition, we carefully study the trade-off between cost savings obtained by continuously migrating tenants and the robustness of servers towards load spikes and failures.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases, parallel databases*

Keywords

Data placement, multi tenancy, in-memory databases, cloud computing, fault tolerance

1. INTRODUCTION

Traditionally, in-memory databases have been employed in performance-sensitive applications such as telephony or financial services markets. In the recent past, however, in-memory databases have become more generally adopted, which is reflected by the product offerings of the major database vendors.[1] At the same time, the Software-as-a-Service (SaaS) model has become increasingly attractive to customers, since it relieves customers of the hassle of operating the system, which entails provisioning the hardware as well as configuring, operating, and maintaining application servers and databases. The SaaS provider, on the other hand, can leverage economies of scale by automating common maintenance tasks as well as by consolidating multiple customers (i.e. tenants) onto the same machine to improve utilization and thereby decrease costs. SAP already operates an in-memory based SaaS application for operational reporting.[2] We expect SaaS applications on in-memory databases to become more widespread in the near future. We focus on read-mostly workloads, which encompass a large class of applications, e.g. SAP ERP, where writes account for less than 10 % of the database workload [17].

In this paper, we consider the problem of consolidating multiple in-memory database tenants onto the smallest possible number of servers to save operating costs. There is a challenging trade-off between low operational cost for the provider and performance as perceived by the customers: only so much consolidation can occur without significant impact on responsiveness. For managing this trade-off, the service provider must address two complementary research challenges, *(i)* resource modeling and *(ii)* data placement. The former entails the estimation of (shared) resource consumption in the presence of multi tenancy on a single server. This is done by characterizing the dominating resources (e.g. CPU, RAM, disk I/O) and quantifying how much each tenant utilizes them. In some cases, these utilization metrics can be mapped to response times produced by a database [7, 21]. This problem has recently been studied for both disk-based databases [5, 18, 7] and in-memory databases [21]. In the latter case, there is no need to model shared disk I/O, which is particularly difficult. The main resources being consumed by in-memory databases, CPU, memory, and bandwidth between CPU and memory, add up mostly linearly [21, 5]. Here, we build on the resource modeling approach presented in [21] and focus on challenge *(ii)*, the assignment of tenants to servers in a way that minimizes the number of required servers (and thus cost). This second challenge has received much less attention, although [5, 18] provide non-linear programs for tenant placement as a first step. Data placement has also been studied for parallel databases [16, 14, 19]. However, in all existing solutions, data placement is done statically, in the sense that diurnal changes in tenant load are not leveraged.

In this paper, we introduce the Robust Tenant Placement and Migration Problem (RTP) and make the case for *incremental* tenant placement, driven by variations in user

[1] http://www.gartner.com/id=2151315

[2] http://www.biondemand.com/businessintelligence

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

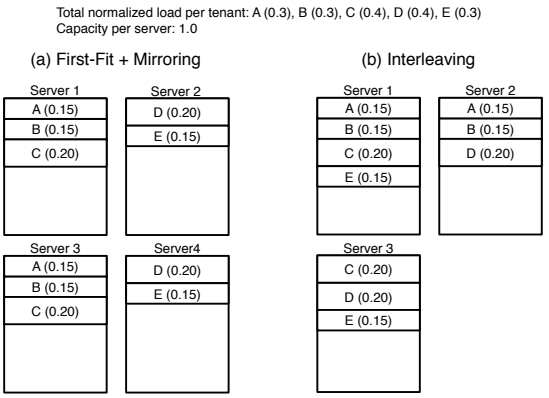


Figure 1: Using interleaved replication for minimizing the number of servers

load. Individual tenant replicas are migrated while the tenant remains on-line [8, 21]. This allows us to make frequent, incremental changes to the tenant placement with the goal of running with the minimal number of servers at each point in time. As our experiments show, incremental placement can decrease server cost for an average business day by up to a factor of ten (measured in Amazon EC2 server hours). More drastic savings can be realized during longer periods of low activity, such as weekends or holidays.

At the core of our approach is the concept of *interleaving* replicas across nodes, which has been studied in the context of fault-tolerance for parallel databases [14]. We are also interested in tolerating server failures, but our problem is different: we try to minimize the number of servers required at each point in time, whereas the existing work on parallel databases assumes a fixed cluster size [16, 14, 19]. Also, a typical SaaS tenant is small and hence there is not much benefit to horizontal partitioning, which is a prerequisite for the existing interleaving strategies [19, 24].

Figure 1 shows an instance of RTP with five tenants and two replicas each, across which all load is shared. A common approach for assigning these tenants to servers is to use a standard first-fit algorithm and mirror the resulting placement (Figure 1a). Note that in this case servers must run at or below half their capacity limit, since the load on a server doubles when its mirror server fails. In contrast, Figure 1b shows an interleaved placement for the same situation. In this case a failure of *server 1* would distribute the load of tenant A and B to *server 2* and the load of tenant C and E to *server 3*. As a result, the layout requires only three instead of four servers while guaranteeing that no server is overloaded upon the failure of any one other server.

In summary, this paper makes the following contributions:

1. We introduce and formalize the Robust Tenant Placement Problem (RTP) including continuous migration, interleaving of replicas, and coping with hardware failure.
2. We present several novel algorithms that solve RTP for both static and incremental scenarios.
3. We provide an extensive experimental evaluation using real-world load traces from a production cloud service run by SAP and benchmark our algorithms in terms of cost, running time, and their ability to maintain response time guarantees in spite of server failures.
4. We develop and evaluate generic over-provisioning strategies for avoiding excess load coming from unexpected load spikes; it turns out that these strategies also help to cope with multiple simultaneous server failures.

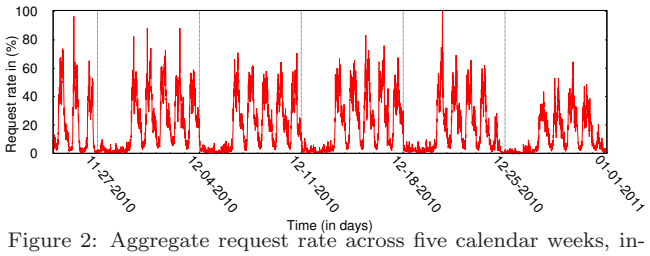


Figure 2: Aggregate request rate across five calendar weeks, including Christmas

Apart from the cost savings possible by using our approach, our experiments may provide guidance to administrators seeking to balance overall operating cost and temporarily overloaded servers after unexpected load spikes. We provide practical advice on the question of how many replicas each tenant should have so that overall cost and robustness towards unexpected load spikes are balanced.

The remainder of this paper is organized as follows. In Section 2, we analyze real-world load traces we received from SAP, motivating the need for placement algorithms that make incremental changes to existing placements at frequent intervals. Section 3 provides a rigorous formalization of RTP. Section 4 contains details on our algorithmic approach. In Section 5, we evaluate our algorithms experimentally. Section 6 surveys related work. Section 7 concludes the paper.

2. ENTERPRISE SaaS OVERVIEW

In order to design more efficient data placement algorithms for multi tenancy, we analyzed real world traces obtained from a production multi-tenant, on-demand application that runs on an in-memory database. These traces were the anonymized application server logs of 100 randomly selected tenants in Europe over a four months period. We also have additional statistics such as the tenants' database size. Figure 2 shows a normalized view of the aggregate number of requests across all tenants over a five week period.[3] In the trace, one can clearly see that tenant behavior follows seasonalities: workdays (with a drop at lunch time), working weeks and annual calendar events. One example for the latter is Christmas Eve which was a Friday in 2010. On this day, the load is considerably lower than for a regular Friday.

Investigating the log data revealed another interesting pattern: a non-negligible number of tenants suddenly appear, use the system actively for 2–3 weeks, then become inactive for a considerable amount of time (say, two weeks), and suddenly become active again for six weeks (Figure 3). Analysis revealed that these were mainly demo and training systems. Capacity planning often neglects those systems. The behavior of tenants in trial periods is particularly hard to predict, which, in part, motivates the incremental placement algorithms presented in this paper.

A more detailed analysis of the load traces can be found in [22]. The following sections leverage these insights for our placement algorithms. Furthermore, we used the real-world traces for evaluating our placement techniques in Section 5.

[3]For privacy reasons, we are not allowed to publish absolute requests rates and the figure only shows relative values.

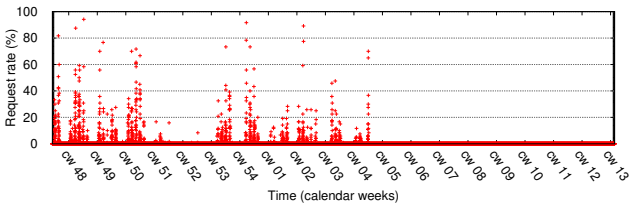


Figure 3: Demo system used only at beginning and end of trial

3. THE ROBUST TENANT PLACEMENT AND MIGRATION PROBLEM

In this section, we formally introduce the problem of assigning tenants to servers such that response time SLOs are met and costs are minimized. We call this problem the *Robust Tenant Placement and Migration Problem (RTP)*.

A valid tenant placement is an assignment of at least two copies of a given number of tenants to a number of (cloud) servers such that

- no server is overloaded in terms of any of its resources
- no server contains more than one copy per tenant, and
- the failure of a single server does not cause overloading any other server.

A tenant t is characterized by its size $\sigma(t)$ (i.e. the amount of space each replica consumes in memory) and its load $\ell(t)$. Our formalization of RTP requires that a workload management technique (i.e. challenge *(i)* in the introduction) is in place to provide $\ell(t)$ as an input. In particular, an estimation of the combined resource consumption of multiple tenants on a machine is required to provide RTP with an indicator for the "fill level" of a server. Therefore, we begin with discussing our assumptions on the underlying resource modeling approach before formalizing RTP.

3.1 Modeling Tenant Resource Requirements

As stated in Section 1, tenant placement requires to estimate the resource consumption and the impact on other clients when a new tenant is placed on a server. This problem has been studied for both disk-based [5, 18, 7], and in-memory databases [21]. In principle, any of these load metrics can be used in RTP. We build on [21], since our focus is also on in-memory databases. This focus considerably simplifies resource modeling in contrast to the disk-based approaches. Schaffner et al. [21] propose a regression model for estimating the combined CPU and CPU-memory bandwidth utilization. The inputs to their regression model are the database sizes and the current request rate of the tenants. All tenants run the same workload. The output is a scalar value representing the combined CPU and CPU-memory bandwidth utilization of the server. This is similar to the approach in [7], where a logical I/O metric is introduced to characterize the dominant resource bottleneck, although their focus is on disk-based systems. The load metric from [21] is additive across multiple tenants. Curino et al. [5] also observe additivity of CPU and memory consumption in disk-based systems. In RTP, we use the load metric from [21] as an input parameter indicating the load of a tenant. In principal, our heuristics (Section 4.1) could also handle non-linear resource models for disk access making our techniques applicable to a wider range of systems.

Service Level Objectives (SLOs). As stated above, tenant consolidation is constrained by compliance with response time SLOs. Here, we require the underlying resource model to incorporate SLOs so that it enables the placement algorithms to guarantee SLO compliance as long as the placement is valid. For providing such SLOs, we may again build on the resource modeling approach in [21] where the load metric is correlated to the query response times of a server in the 99-th percentile (across all tenants and for the given workload). Normalizing the maximum load prior to an SLO violation to 1.0 allows us to express SLO compliance as $\sum_{t \in T} \ell(t) \leq 1.0$, where $\ell(\cdot)$ is the *load* of a tenant in T, the set of all tenants on a server. Thus, as long as the placement algorithm keeps the total load of a server below 1.0, the response time goals will be met (similar to [18]).

3.2 Formalization

Based on the above considerations, RTP can be described as follows. Given an initial placement (potentially containing overloaded servers), find a valid placement by migrating not more than a limited amount of data (called the *migration budget*) such that the number of *active* servers is minimal. We call a server active if it contains at least one tenant. We consider a server overloaded when one of its resources is used beyond its capacity limit. This includes additional load redirected to a server when another server has failed.

A valid instance of RTP has the following data as **input**:

- $T \subseteq \mathbb{N}$, the set of tenants.
- $N \subseteq \mathbb{N}$, the set of available servers.
- $R = \{1, 2, \ldots, r(t)\}$, the replicas per tenant. $r(t) \geq 2$ is the (fixed) number of replicas per tenant; Section 3.4 contains details on how to obtain $r(t)$.
- $\sigma : T \to \mathbb{N}^+$, a function returning the DRAM requirement of a given tenant.
- $\text{cap}_\sigma : N \to \mathbb{N}^+$, a function returning the DRAM capacity of a given server.
- $\ell : T \to \mathbb{Q}^+$, a function returning the current load of a given tenant. As discussed in Section 3.1, a single metric is sufficient given our focus on in-memory databases.
- $\text{cap}_\ell : N \to \mathbb{Q}^+$, a function returning the request processing capacity of a given server.
- An existing tenant placement.
- A migration budget $\delta \in \mathbb{N}$. This parameter depends on the length of a reorganization interval, i.e. the time after which a placement is reconsidered.
- $\nu_i : N \to [0, 1]$, a function returning the capacity loss when a server is a migration target.
- $\mu_i : N \to [0, 1]$, a function returning the capacity loss when a server is a migration source. We will discuss how to obtain ν and μ later in this section.

In an assignment formulation of RTP, a valid solution must assign appropriate values to the following decision variables as **output**:

- a binary decision variable $y \in \{0, 1\}^{N \times T \times R}$ with $y_{t,i}^{(k)} = 1$ if and only if copy k of tenant t is on server i.
- $s \in \{0, 1\}^N$, where $s_i = 1$ denotes that server i is active.
- $p \in \mathbb{Q}_+^N$, where p_i denotes the capacity of server i that must be left unused such that additional load due to a single server failure does not cause an SLO violation. We call p_i the *penalty* that must be reserved on server i. As stated in Section 3.1, an SLO violation occurs when the total load of any non-failed server exceeds 1.0 as a consequence of another server failing.

The objective of RTP is to minimize the number of active servers, i.e. $\min \sum_{i \in N} s_i$.

A solution of RTP must obey the following constraints.

$$\sum_{i \in N} y_{t,i}^{(k)} = 1 \qquad \forall t \in T,\ \forall k \in R \tag{1}$$

Constraint (1) ensures that each replica $1 \leq k \leq r(t)$ of a tenant t is assigned to a server exactly once.

$$\sum_{k \in R} y_{t,i}^{(k)} \leq 1 \qquad \forall t \in T,\ \forall i \in N \tag{2}$$

Constraint (2) ensures that no two copies of the same tenant are placed on the same server.

$$\sum_{t \in T} \sum_{k \in R} \sigma(t) \cdot y_{t,i}^{(k)} \leq \mathrm{cap}_\sigma(i) \cdot s_i \qquad \forall i \in N \tag{3}$$

Constraint (3) ensures that the total size of all tenants on a server does not exceed the server's DRAM capacity. If at least one tenant is assigned to the server, s_i is set to one.

Similarly, the following constraint ensures that the total load of all tenants on a server does not exceed the processing capabilities of the server. We assume that load can be shared across replicas: each server holding a replica of tenant t receives only $1/r(t)$-th of $\ell(t)$. We will justify this assumption in Section 3.3.

$$\sum_{t \in T} \sum_{k \in R} \frac{\ell(t)}{r(t)} \cdot y_{t,i}^{(k)} + p_i \leq \nu_i \cdot \mathrm{cap}_\ell(i) \cdot s_i \ \forall i \in N\,. \tag{4}$$

Each server must be capable of potentially handling additional load in case another server fails. The spare capacity reserved for this excess load is captured by a penalty p_i in Constraint (4). The following constraint defines the penalty.

$$p_i = \max_{j \in N : j \neq i} \sum_{t \in T} \sum_{k,k' \in R} \frac{\ell(t)}{r(t)^2 - r(t)} y_{t,i}^{(k)} y_{t,j}^{(k')} \qquad \forall i \in N \tag{5}$$

What fraction of a tenant's load must be added to p_i depends on the number of remaining replicas. If server j handled a fraction $\ell(t)/r(t)$ of the load of tenant t load prior to the failure, then the remaining $r(t) - 1$ replicas of tenant t must share the load after the failure. Hence, the extra load that server i must support is

$$\frac{\ell(t)}{r(t)} \frac{1}{r(t) - 1} = \frac{\ell(t)}{r(t)^2 - r(t)}\,. \tag{6}$$

Constraint (5) ensures that p_i is set large enough to guarantee that even the failure of the "worst case" other server $j \neq i$ would not result in overloading server i. The constraint renders heuristics for bin-packing unusable for RTP: given three servers U, V, and W, moving a tenant from V to W may increase p_U and thus render server U unable to sustain the extra load coming from another server failing.

RTP guarantees that performance SLOs are met while tenants are being migrated. The intuition that migration affects query latencies is quantified in [21] for those servers participating in migrations. During a migration, a source server with a total load of $\mu = 0.85$ (or a destination server with a total load of $\nu = 0.82$) produces a response time of one second in the 99-th percentile.[4] We build on this in the following Constraints. Note that Constraint (4) implicitly takes the given placement into account via the parameter ν_i: whenever server i is a migration target, i.e. a tenant is assigned to server i that was not assigned to this server previously, the load capacity of the server drops by a factor of $\nu_i < 1$. If i is not a migration target we have $\nu_i = 1$.

For notational convenience, we define $T_{\text{mig}} := \{t \in T : \text{a copy of } t \text{ was moved}\}$.

[4]This assumes that all migrations are executed sequentially. Degradation factors depend on the workload, the hardware, and on how migrations are implemented in the DBMS.

$$\forall\, t \in T_{\text{mig}}\ \exists\, i \in N\ :$$

$$\sum_{t' \in T} \sum_{k \in R} \frac{\ell(t')}{r(t)} \cdot y_{t',i}^{(k)} + p_i \leq \mu \cdot \mathrm{cap}_\ell(i) \cdot s_i \quad . \tag{7}$$

Constraint (7) ensures that for every tenant being migrated, a server exists that has enough spare capacity to act as a migration source. A stronger version of (7), in which we replace T_{mig} by T, guarantees that every tenant has a replica on a server that may act as a migration source.

Constraint (8) enforces the migration budget δ.

$$\sum_{t \in T_{\text{mig}}} \sigma(t) \leq \delta \tag{8}$$

Constraints (7) and (8) may render RTP infeasible in case of extreme load change in comparison to the given placement. In such cases, it may occur that *(i)* no server can act as a safe migration source for a tenant or *(ii)* the migration budget is not large enough for repairing all overloaded servers. When an infeasibility occurs, it becomes necessary to tolerate SLO violations while restoring a valid and flexible placement. Besides temporarily dropping constraints, a change in the objective function becomes necessary to minimize SLO violations. Instead of minimizing the number of active servers, a placement should be found with the lowest number of overloaded servers, which can be formalized as follows. We introduce a variable $e \in \mathbb{Q}_+^N$ which measures the excess load on a server. For $i \in N$, we define

$$e_i = \sum_{t \in T} \sum_{k \in R} \frac{\ell(t)}{r(t)} \cdot y_{t,i}^{(k)} + p_i - \mathrm{cap}_\ell(i) \tag{9}$$

and alternative objective functions are

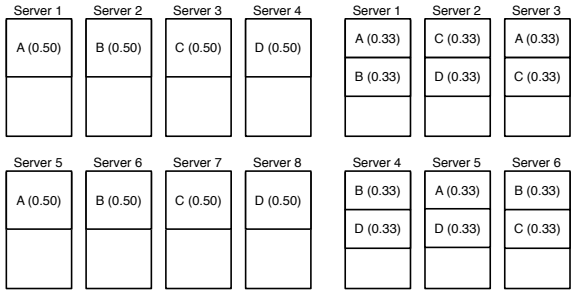
$$\min \sum_{i \in N} e_i \quad \text{or} \quad \min \max_{i \in N} e_i. \tag{10}$$

Computational Complexity. We introduce a special case of incremental RTP, which will be useful in our experiments as well as for discussing the (computational) complexity of RTP. We call the subclass of incremental RTP where *(i)* no initial placement is given, *(ii)* both $\nu_i = \mu_i = 1$ for all i, and *(iii)* $\delta = \infty$, *static* RTP. Note that an optimal solution of static RTP is a lower bound for optimal solutions of incremental RTP. A reduction from the `PARTITION` problem [11] shows the (weak) $\mathcal{NP}$-hardness of static RTP. Consequently, for an arbitrary migration budget, incremental RTP is also $\mathcal{NP}$-hard. We omit proofs due to space restrictions.

3.3 Load Distribution Across Replicas

Our formulation of RTP assumes load to be distributed equally among a tenant's replicas (e.g. in Constraint (4)). This allows to serve more requests per tenant. However, we can only obtain this benefit when the workload has read-mostly characteristics. This applies to real-time analytical database applications: an analysis of several enterprise database workloads showed that write queries account for less than 10 % of the total workload [17].

To simplify the presentation of our formal model, Constraint (4) assumes a read-only workload. Splitting load into weighted read and write components, (4) could be modified such that our formulation of RTP is independent of the workload characteristics. For a write-mostly workload, however, the load cannot be split among replicas; instead, all replicas are exposed to the full load (assuming that writes go to all replicas). Also, no load-redistribution occurs in case of a failure. All other aspects of the problem formulation remain intact with write-intensive workloads. However, an



(a) $r(t) = 2, \forall t \in T$ (b) $r(t) = 3, \forall t \in T$

Figure 4: Required number of servers dependent on $r(t)$

exhaustive study of the impact of writes on replicated tenant placement is beyond the scope of this paper.

3.4 Choosing the Number of Replicas

In the previous section, the number of replicas per tenant $r(t)$ was treated as an input parameter to our optimization problem. In the following, we discuss how to obtain $r(t)$. Intuition suggests to set $r(t)$ as low as possible, since *(i)* more replicas require more space, which could lead to a higher number of active servers, and *(ii)* the problem becomes more constrained. However, increasing the number of replicas beyond $r(t) = 2$ becomes necessary when the load of a tenant is so high that a single server cannot handle half of it. The number of copies $r(t)$ of a tenant t must be chosen such that $\ell(t)/r(t) < \text{cap}_\ell(i)$. In addition, server i must be able to handle the extra load coming from another server failing that also holds a copy of t. Hence, we must choose $r(t)$ in such a way that the following inequality applies.

$$\frac{\ell(t)}{r(t)} + \frac{\ell(t)}{r(t)^2 - r(t)} \leq \text{cap}_\ell(i) \qquad \forall i \in N \qquad (11)$$

We rearrange Inequality (11) for $r(t)$ to obtain:

$$r(t) := \max(2, \left\lceil \frac{\ell(t)}{\text{cap}_\ell(t)} + 1 \right\rceil) \qquad (12)$$

In contrast to our intuition, increasing the number of replicas beyond the lower bound as defined in (12) can lead to placements with fewer servers, as shown in Example 1.

Example 1. *Consider four tenants A to D, each with a load* 1.0 *and servers with capacity* 1.0. *For two replicas per tenant, as shown in Figure 4a, eight servers are necessary to place all tenants. The load on all servers including spare capacity reserved to accommodate potential server failures (i.e. p_i) is 1.0. If we allow three replicas per tenant, as shown in Figure 4b, then a total of six servers are sufficient. Also in this case, the load on all servers including p_i is 1.0.*

4. ALGORITHMS FOR RTP

In this section, we present algorithms for solving RTP. We start by discussing heuristics for static RTP because they form the foundation for our incremental algorithms. We also tackle RTP with exact algorithms, in particular, with mixed integer programming (MIP) solvers. The challenge lies in linearizing the non-linear constraints to obtain a MIP formulation. Powerful solvers like CPLEX can then be used to provide solutions and bounds on the optimal solution.[5]

[5]http://www.ilog.com/products/cplex

Due to space restrictions, we omit the linearizations and focus on the heuristics.

4.1 Heuristics for Static RTP

Greedy Heuristics. For the related bin-packing problem, greedy heuristics deliver good results [11]. Another reason for considering greedy variants are their speed. Even for short migration intervals, a greedy heuristic can be used when more complex algorithms are prohibitively slow.

Our greedy algorithms are loosely based on the well-known best-fit algorithm [4]. When placing a single replica of a tenant, for each server its total load including its *penalty* (Section 3.2) is computed. The servers are then ordered according to load plus penalty in decreasing order. Similar to best-fit, the first server that has enough free capacity is selected. If no active server has enough capacity, then the tenant is placed on a new server. Apart from load plus penalty on the servers, we consider Constraints (1)–(7).

This basic mechanism for placing a single replica of a tenant is called *robustfit-single-replica*. It is the basis for the algorithms robustfit-s-mirror and robustfit-s-interl., which will now be discussed. robustfit-s-mirror first sorts all tenants by load in descending order and places the first replica of each tenant. Since there is no penalty when there is only one copy, the algorithm assumes a server capacity of $\frac{\mu \cdot \text{cap}_\ell(i)}{2}$ in this step. Then, all servers are mirrored. Finally, the algorithm places additional replicas individually for tenants that require more than two replicas (see Section 3.4). robustfit-s-interl. also sorts all tenants and then, tenant after tenant, places all replicas of each tenant. For the first replica of each tenant, a server capacity of $\mu \cdot \text{cap}_\ell(i)$ is assumed. For all other replicas the algorithm assumes a capacity of $\text{cap}_\ell(i)$. This results in a placement where each tenant has a safe source server. Also, tenant replicas are naturally interleaved across servers. Both algorithms have polynomial complexity and run fast for the problem sizes we consider in this paper.

Metaheuristic: Tabu Search. Having considered fast greedy heuristics, we consider a computationally more expensive heuristic next. We propose an adaptation of Tabu search [12], used as an improvement heuristic. Given a starting solution (e.g. obtained by one of the greedy heuristics above), tabu-static tries to remove an active server S by traversing the search space as follows. Every valid solution of RTP is a point in the search space. We move from one valid solution to another valid solution by migrating a tenant t from S to a different server T, even if this move leads to an invalid placement. Next, we fix possible conflicts (if possible without placing a tenant on S). In order to avoid both cycling and stalling in a local optimum, a so-called Tabu list stores each move (t, S, T). We only allow a move if it is not in the Tabu list. When the list reaches a certain length, the oldest element is removed. The search aborts if, after a certain number of iterations, no placement was found that does not use S. If a solution without S was found, search continues from the new solution with the goal of removing another server. The performance of tabu-static relies on the careful adjustment of its parameters (e.g. Tabu list length, choice of server to be cleared out, order in which tenants are moved). We identified good settings using careful experimentation.

4.2 Algorithms for Incremental RTP

The static algorithms discussed above are the basis for our incremental placement heuristics. In order to leverage

the different heuristics, we use a metaheuristic, which acts as a framework for all incremental placement strategies. Its main benefit is a significant reduction of the solution search space, leading to lower overall algorithm execution times.

4.2.1 A Framework for Incremental RTP

The framework consists of six phases. They are executed at the beginning of each reorganization interval, independent of the algorithm that is currently run. Individual algorithms must plug in a method for placing a single replica of a tenant or replace entire phases. Such a method is for example the robustfit-single-replica method described above. An incremental algorithm can also provide an own implementation for individual phases of the framework. The six phases of this framework are as follows.

1. Delete unnecessary replicas. When the load of a tenant has decreased in comparison to the previous interval, it might be the case that removing a replica of the tenant is possible (see also the discussion on the lower bound on the number of replicas in Section 3.4). Therefore, in this phase, a heuristically selected replica of all tenants meeting this condition is deleted. Note that deleting a tenant does not count towards the migration budget.

2. Ensure migration flexibility. This phase ensures that all tenants have at least one replica on a server that has enough spare capacity to participate in a migration as a source server (Constraint (7)). For determining this server, the plugged-in algorithm is used. This results in the ability to migrate tenants without causing SLO violations.

3. Create missing replicas. This phase handles the opposite case of phase (1), where the lower bound on a tenant's replicas has increased as a result of increasing load. The plugged-in algorithm is used to place enough extra replicas as necessary to match the new lower bound.

4. Fix overloaded servers. This phase repairs overloaded servers by moving tenants away from them until they are no longer overloaded. The plugged-in algorithm is used to determine the target servers for replicas that are moved.

5. Reduce number of active servers. All servers are ordered by total load plus penalty. Then, all tenants on the most lightly loaded server are moved to other servers using the plugged-in algorithm. This phase is repeated with the next server up to the point where the server cannot be emptied without creating a new server.

6. Minimize maximum load. When a reduction of the number of servers is no longer possible, this phase flattens out the variance in load plus penalty across all servers. The goal is to avoid having servers in the placement that have a much higher penalty than other servers. Again, the plugged-in heuristic is used. This phase terminates when the migration budget is exhausted or further migrations would have too small an effect on the variance.

The execution of the framework is immediately aborted when the migration budget is exhausted. When too low a value for the migration budget is chosen, the placement may be invalid (i.e. it does not satisfy all the constraints of RTP) after premature termination. A placement is always valid after completion of phase (4).

Note that the execution order of the above framework is itself a heuristic. Experimentation has revealed that executing phase (4) after phase (2) results in fewer servers than the inverse order, because some overloaded servers are repaired as a side product of finding a safe migration source for the tenants. Note further that the question of deciding how many replicas a tenant should have is orthogonal to this framework. Similar to plug-in methods for placing individual replicas, different strategies for determining the replication factor can be plugged in. The standard method is to use exactly as many replicas as suggested by the lower bound. Another method is to increase the lower bound by a fixed offset. A more sophisticated method is to set the number of replicas across all tenants in a way that all replicas receive more or less the same load. A last method is to repair overloaded servers in phase (4) by creating additional replicas elsewhere, thus decreasing the load of the tenant on the overloaded server. In the following, we discuss the plug-in algorithms that we have developed for this framework.

4.2.2 Greedy Heuristics

The simplest and fastest algorithm is called robustfit-inc. and merely entails the method for placing a single replica using robustfit-single-replica (described in Section 4.1). This method is plugged into the above framework as is. Since the space of possible actions when transforming a given placement into a new placement is very large, we created splitmerge-inc. This algorithm acts exactly as robustfit-inc. but provides an own implementation of phases (4) and (5) in the framework above. In phase (4) the only allowed operation is splitting each overloaded server into two servers. In phase (5), conversely, merging two servers into one is the only legal operation, although multiple server pairs can be merged in one step. Since the underlying robustfit-single-replica method is very fast, we use a more complex procedure for deciding what servers to merge: splitmerge-inc. builds up its list of merge pairs by checking whether two servers U and V can be merged for all candidate pairs $U \times V$. The method in splitmerge-inc. for removing servers is effective but computationally intensive. Its approach for fixing overloaded servers is rather simple: overloaded servers cannot be fixed without creating one new server per overloaded server, which seems too drastic. We therefore replaced splitmerge-inc.'s implementation of phase (4) with the standard one again and used robustfit-single-replica as the plug-in heuristic referring to this as robustfit-merge.

4.2.3 Metaheuristic: Tabu Search

We also use our Tabu search for incremental RTP: tabu-inc. replaces phase (5) with the Tabu search from Section 4.1. Note that tabu-inc. does not use a solution obtained by a greedy heuristic as the starting solution; it simply starts with the given placement. tabu-inc. simply omits phase (6), which saves some of the migration budget and thereby allows the Tabu search to run longer. The next heuristic, tabu-inc.-long works exactly as tabu-inc., except that the parameters of the Tabu search are set in such a way that it runs significantly longer (and thus visits more solutions). Finally, we combine robustfit-inc. with tabu-inc. into tabu-robustfit. This algorithm runs robustfit-inc. as a preprocessing step, thereby omitting phase (6) so that the remaining migration budget can be used to improve the solution using Tabu search. Another variant of this algorithm is tabu-robustfit-l., where the Tabu component runs longer.

4.2.4 Portfolio Approach

The portfolio approach combines all heuristics for incremental RTP. We simply run all heuristics starting from the

same, best-known solution. We then pick the best solution among all algorithms as the next solution. Choosing the best solution as the next solution is itself a heuristic approach.

5. EXPERIMENTS

In this section, we evaluate our algorithms for RTP. Our evaluation is based on real-world load traces, which we obtained from a production cluster of the aforementioned SAP application. To preserve customer privacy, an anonymized random sample of 100 tenants was given to us. While this is only a fraction of the customer base, the sample provides a realistic profile of the whole customer base. Unfortunately, the sample is not large enough for experiments at scale. We thus used the technique presented in [22] to bootstrap new tenants. The newly created tenants have load traces similar to the original tenants and follow the same periodical patterns. Our final testing dataset contains 435 tenants.

The evaluation is structured as follows: Section 5.1 discusses the performance of our algorithms w. r. t. their balance among *(i)* the number of active servers, *(ii)* computation time, and *(iii)* their robustness towards load changes. We will see that robustfit-inc. achieves a good balance between all three measures. Consequently, Section 5.2 explores robustfit-inc. further. We investigate lower bounds for server cost. We also study the effects of increasing the number of replicas per tenant beyond the minimum, which has interesting effects on the number of active servers and the stability of a placement over the day. In Section 5.3 we study *generic over-provisioning strategies* to reduce the impact of temporarily overloaded servers until it becomes negligible.

In all experiments, we assume that servers have a DRAM capacity of $\text{cap}_\sigma = 32\,\text{GB}$. We further assume a homogeneous server load capacity of $\text{cap}_\ell = 1.0$. While our algorithms also work for heterogeneous servers, homogeneity simplifies the presentation of our experiments. Results from the literature suggest that approximability results for homogenous servers will carry over to heterogenous servers [2, 15]. In the experiments, we set the migration budget to $\delta = 27\,\text{GB}$ because such an amount can safely be migrated in a ten minute interval using SAP's in-memory database and a 10 Gbit Ethernet interconnect [21]. Our experiments were conducted on an Intel Xeon X7560 server with 2.27 GHz running Linux. We implemented our heuristics in Scala and used Cplex as a MIP solver. We have not yet parallelized our heuristics (in contrast to Cplex). We would expect a significant speed-up from a multi-threaded implementation, especially for Tabu search and portfolio.

5.1 Comparison of Heuristics for RTP

In order to evaluate our heuristics for solving (incremental) RTP we consider the following three measures:

1. the *cost* associated with the resulting placements,[6]
2. the *computation times* required by the algorithms, and
3. *robustness* of the placement towards unexpected increases in tenant load.

Not all can be optimized for at the same time; consequently, a trade-off between these measures must be found. A particularly inexpensive placement may require an unrealistic amount of computing time and then, at the same time, the tenants might be packed so tightly that servers are prone to temporary overloads when load changes.

Experiment (i): *Incremental Placement vs. State of the Art.* We compare our incremental heuristics against two baselines published in [25] and [5]. Table 1 summarizes the benefits of our incremental algorithms over these two static approaches measured on a typical working day. To allow a fair comparison, we modified both baseline approaches such that they also encompass the replication, load balancing and failure-robustness properties of RTP.

The first static approach, modeled after [25], entails monitoring all tenants for one week and observing the peak load of each tenant within that period. Afterwards, one provisions for this peak load. We used the week directly preceding the Wednesday chosen for our experiments to estimate the maximum load for each tenant. We then solved static RTP for the observed peak loads using greedy heuristics. Table 1 shows that robustfit-s-mirror, the simplest static algorithm, requires 320 servers, whereas robustfit-s-interl., the best static algorithm in this case, requires 192 servers.

The second static approach, kairos-MIP, modeled after [5], also entails monitoring all tenants for a period of time and then computing a static placement. In contrast to [25], where this placement is computed based on the maximum load requirements observed for all tenants during the observation period, kairos-MIP tries to consolidate more aggressively by requiring its (static) placement to be valid across all ten minute intervals in the observation. We picked the Wednesday of the week preceding our exemplary Wednesday and tried to compute a placement with kairos-MIP, our implementation of [5] in Cplex. Note that given our focus on in-memory databases and the absence of shared disk access, we can use a MIP formulation, which has computational advantages. However, the corresponding MIP formulation becomes so large for our trace data that kairos-MIP is computationally unsolvable within one week. We therefore picked a subset of high-load ten minute intervals from the Wednesday and ran kairos-MIP on this subset. Hence, we only obtain a lower bound on the actual cost of the kairos-MIP placement; we can expect the kairos-MIP placement over all ten minute intervals (if it was computable) to be significantly more costly. Our experimentation with smaller sets of tenants and servers suggests that working on a subset of ten minute intervals results in placements which are approx. 60 % cheaper than including all ten minute intervals. Table 1 shows that kairos-MIP requires (at least) 45 servers. Note that the running time for kairos-MIP even when run on a subset of ten minute intervals is much higher than reported in [5], although we use Cplex. This may be partially due to the additional constraints of RTP that we added for comparability (e.g. Constraint (5)). However, the main reason is probably the much higher number of tenants and servers in our experimental data compared to [5].

In contrast to both static baselines, our incremental algorithms, which alter the placement in ten minute intervals, require between 33 and 40 servers during times of peak load and much fewer servers during the night and times of low load (e.g. weekends). Table 1 shows the cost for server rent for all incremental algorithms. On average, the cost for server rent is an order of magnitude lower when using an incremental algorithm as opposed to using static provisioning based on peak load while being computationally comparable. Incremental placement is still a factor of 2.2 cheaper

[6]We measure operational cost as accrued when using a varying number of "high memory" instances on Amazon EC2. See http://aws.amazon.com/ec2/pricing/.

than the lower bound on the Kairos approach, while being far superior in terms of computational times.

Table 1: Server cost and running time of heuristics for RTP

Algorithm	**Cost**	**Servers**	**Running time**	
		max	*avg*	*max*
Static:				
robustfit-s-mirror	$3456.00	320		66.4 s
robustfit-s-interl.	$2073.60	192		481.3 s
kairos-MIP	> $432.00	> 45		> 3 days
Incremental:				
tabu-inc.	$273.83	40	2.5 s	5.9 s
tabu-inc.-long	$208.20	34	26.8 s	87.0 s
tabu-robustfit	$202.95	33	3.0 s	10.8 s
robustfit-inc.	$201.45	39	1.7 s	3.7 s
splitmerge-inc.	$200.18	38	95.5 s	321.6 s
robustfit-merge	$198.08	32	84.2 s	256.4 s
tabu-robustfit-l.	$193.05	33	19.8 s	60.5 s
portfolio	$191.55	33	182.1 s	565.3 s

Note that for incremental placement there might be overheads associated with shutting down and powering up different nodes dynamically in a cluster. Prior to shutting down a node, all tenants must be replicated away from the node. RTP ensures that this is done without SLO violations. Also, copying tenants to other nodes counts towards the migration budget. Powering up a node incurs some time for provisioning, which is not modeled in RTP. A reasonably-priced workaround is to maintain a pool of 2–5 spare nodes, which can instantly be filled with tenants when required.

Experiment (ii): *Incremental Algorithms: Cost vs. Running time.* The time in a ten minute interval is split into the time for algorithmic computation and the remainder, which is used to physically carry out the migrations. The shorter the running time of an algorithm the more time is available for performing migrations. A short running time also indicates good scalability of an algorithm towards larger problem sizes. Among the fast algorithms with an average running time below 10 s (see Table 1), robustfit-inc. finds the solutions with the lowest cost. It is also the fastest algorithm overall, and thus the best option for short reorganization intervals. Among the longer-running heuristics, portfolio naturally delivers the best results because it combines all other incremental heuristics and selects the placement with the fewest servers in each ten minute interval. portfolio is also by far the slowest (incremental) algorithm. tabu-robustfit-l. is almost as good as portfolio w. r. t. server cost. However, on average, tabu-robustfit-l. is more than ten times faster than portfolio. tabu-robustfit-l. is the best choice if one can allow investing up to one minute of computation per ten minute interval. At certain times during the day portfolio produces placements requiring more servers than some of the other incremental heuristics (e.g. robustfit-merge). This behavior results in portfolio requiring a higher maximum number of servers than robustfit-merge (see Table 1). This phenomenon—counter-intuitive at first since portfolio is supposedly the best incremental heuristic—highlights the strong influence that the given placement from the previous interval has on the ability of any incremental algorithm to minimize the number of active servers.

Experiment (iii): *Robustness Towards Load Spikes.* When using an incremental placement strategy, one tries to find a placement using the minimal number of servers while still providing just enough resources to handle the load of all tenants without violating response time SLOs. This results in situations where servers have little spare capacity. When changes in tenant load are observed, a new placement is computed and tenants are migrated away from overloaded servers. When using a static placement strategy, in contrast, all servers must have enough spare capacity to handle an estimated peak load over a longer time period. In the following, we study how many servers are *temporarily overloaded* in each ten minute interval when an incremental placement strategy is used. Here, a temporarily overloaded server has a load beyond its load capacity limit at the beginning of a ten minute interval, i.e. after new values for the load of the tenants have been observed and before a new incremental placement is computed and put in place. This metric is an indicator for a placement's robustness towards unexpected load spikes. The fact that servers are temporarily overloaded while the placement is being re-organized in response to a load spike is perhaps the most important downside of incremental placement. Managing the trade-off between temporary overloads and cost for server rent is a key challenge.

Figure 5a provides two main insights. Firstly, temporary overloads occur mostly in the morning when people come in to work. This is the time when load increases drastically between adjacent ten minute intervals. Secondly, temporary overloads affect a large fraction of all active servers. The latter is actually positive: due to interleaving, excess load is distributed across many servers, which avoids local hotspots in the cluster. As can be seen in Figure 5b, the average excess load on all servers is moderate for most ten minute intervals. The largest component of the average excess load on the servers is due to headroom that is reserved for server failures (i.e. our penalty). Figure 5c shows the net average overload across all servers, without including the penalty in the load on the servers. For our exemplary business day, as long as no server failure occurs exactly at 7:20 a.m., temporarily overloaded servers actually never exceed their capacity limit by more than 10%.

Surprisingly, placements computed with the splitmerge-inc. algorithm are extremely robust: only 30% of the servers are beyond their load capacity when considering penalty, and no server at all is overloaded when considering only the actual load without penalty. In the latter case there is not a single ten minute interval with an overloaded server. Although splitmerge-inc. is clearly superior to our other heuristics in this regard, its high running times (between 1.5 and 5.5 minutes per ten minute interval) make its use mostly impracticable. Note also that splitmerge-inc. is harder to parallelize than for example tabu-inc. or portfolio due to its complex merge phase. Based on this experiment, it becomes clear that our heuristics must be extended to minimize the impact of temporarily overloaded servers. We develop and evaluate appropriate techniques in Section 5.3.

We conclude that robustfit-inc. provides the best balance between server cost, running time, and robustness towards temporary load spikes. The overall most robust algorithm, splitmerge-inc., has prohibitively long running times. Therefore, we evaluate robustfit-inc. in more detail in the following. Due to space restrictions, we omit experiments with a varying migration budget in this paper. It turns out that tabu-inc. has some characteristics that make it favorable over robustfit-inc. for small migration budgets and thus shorter reorganization intervals. However, reorganization in-

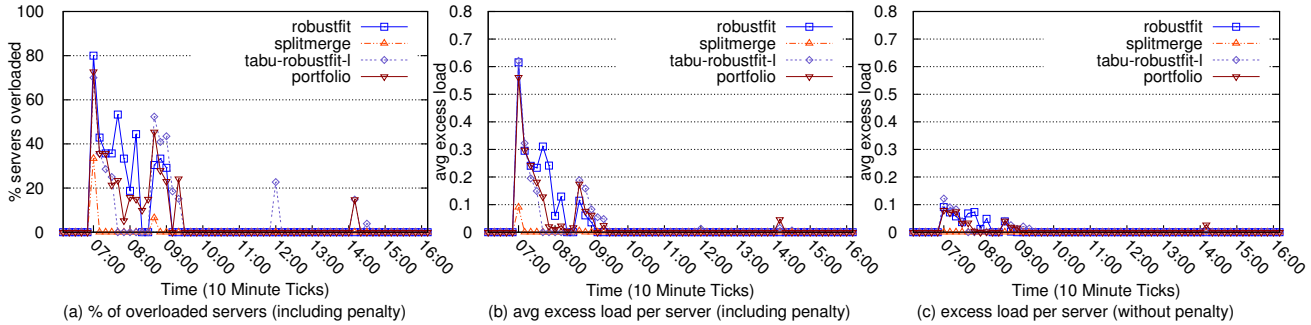


Figure 5: Overloaded servers during working hours for selected algorithms

tervals shorter than ten minutes could result in "thrashing" in the sense of overreacting to short-lived load bursts.

5.2 Advanced Experiments with Robustfit

Having established that immense cost savings can be realized using incremental placement strategies, a natural next question is by how much our algorithms deviate from placements which are optimal in terms of the number of required servers. We therefore investigate lower bounds on server cost in this section. We also consider varying the number of replicas per tenants beyond the minimum.

Experiment (iv): *Lower Bounds on Operational Cost.* In the following, we compare robustfit-inc. with two sets of cost baselines. The first comes from solving the static variant of RTP with our heuristics. The second comes from running Cplex on our MIP formulation for RTP, with the goal of finding *optimal* solutions. Both lower bounds are only of theoretical interest because *(i)* running a static algorithm in each ten minute interval ignores migration costs and *(ii)* we allow a time limit of three hours per ten minute interval for Cplex. Table 2 contains details.

Table 2: Gap between incremental solutions and lower bounds

Incremental Algorithm	Lower Bound	Gap in % avg	Gap in % max
robustfit-inc.	tabu-static-long	8	63
robustfit-inc.	RTP-MIP	19	75
RTP-MIP	RTP-MIP-lower-bound	17	42

Surprisingly, robustfit-inc. performs almost good as tabu-static-long (the best static heuristic) on average, even though the incremental placement problem intuitively seems more challenging than the static one. The large maximum gap is in fact due to an outlier: the second largest gap is 40 %. For more than 30 out of 144 ten minute intervals, robustfit-inc. even requires fewer servers than tabu-static-long or other static algorithms. One reason for the good performance of robustfit-inc. might be that it often has the opportunity to start from a good solution obtained in the previous ten minute interval. Incremental improvements of good solutions are carried forward by robustfit-inc.

Heuristics for static RTP provide an empirical lower bound on the required number of servers attainable in a timespan proportional to the problem size. However, exploring all combinatorial options systematically may lead to solutions with fewer servers. We therefore report on solving MIP formulations of RTP with Cplex next. We are interested in studying the relative gap between the solutions obtained by Cplex and robustfit-inc. Unfortunately, the standard problem size used in our experiments is too large for experimentation with Cplex. We therefore use a smaller set of tenants (136 tenants) on which we run both Cplex and robustfit-inc.

Cplex can often improve our heuristic solutions, sometimes by a considerable amount. The low average gap (see lower part of Table 2) however clearly speaks for robustfit-inc. While the results here have been obtained based on the MIP formulation for RTP, we observed similar results when running Cplex on the static variant with a 24 hour time limit per ten minute interval. Based on these experiments, we conjecture the relative gap between exact solutions and heuristically obtained solutions to be similar for larger problem sizes (as is the case for the related bin-packing problem [11]). Cplex also computes lower bounds on the optimal solution (see last line of Table 2). The gap between the best Cplex solution and this lower bound does not necessarily indicate that the best Cplex solution can actually be further improved, but rather that the lower bounds are weak. This is another similarity to the bin-packing problem where an assignment formulation leads to lower bounds relatively far below the actual optimum [23]. We conclude that the placements obtained by robustfit-inc. are close enough to the theoretical optimum, especially considering its speed.

Experiment (v): *Varying the Number of Tenant Replicas.* In all previous experiments the number of replicas per tenant was set to the minimum with (12). Section 4.2.1 listed several approaches for dynamically computing the number of replicas. In this experiment, we evaluate the simplest one: varying the number of replicas per tenant by adding an offset between one and five to the minimum number of replicas. Figure 6 shows that a higher replication factor decreases the variance in the active number of servers over the day. Tenant size becomes the dominant resource dimension as the number of replicas increases, up to a point where tenant load is no longer the limiting factor. To our surprise we found that the maximum number of servers required during peak load decreases drastically as the offset increases. Conversely, during times of low load, a high offset increases the number of active servers. For our Wednesday, an offset of four is best during peak load and an offset of zero is best during the period where load is at its lowest level. There are stages in between where offsets of two and three do best. At peak load, an offset of five results in the smallest number of servers. Table 3 shows that cost for server rent does not increase monotonically with a higher replication offset. In fact, increasing the offset from zero to one decreases cost from

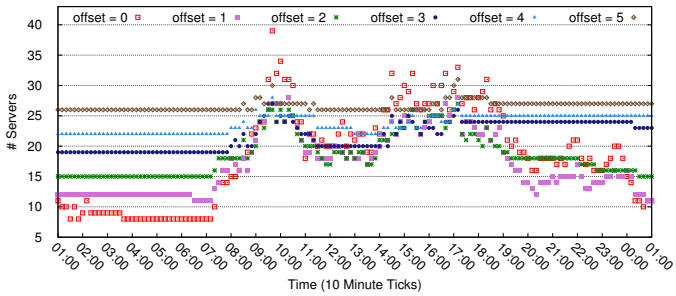


Figure 6: Number of active servers on a typical day with a varying replication factor

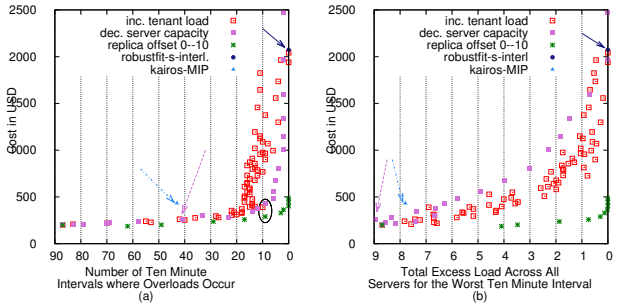


Figure 7: Performance of different generic over-provisioning strategies for avoiding overloaded servers

$201 to $187. As a point of comparison, running portfolio—supposedly the best incremental heuristic—with an offset of zero accounts for a daily cost of $192 (see Table 1).

Table 3: Daily server cost with varying offset

Offset	0	1	2	3	4	5
Cost ($)	201	186	201	237	257	289
Servers (max)	39	28	27	27	28	31

Dynamically varying the number of replicas over the day is a promising avenue for future work.

5.3 Generic Over-Provisioning Strategies

As stated in Experiment 5.1, incremental placement requires trading off cost and robustness towards load spikes. In the following, we consider generic measures for reducing the number of temporarily overloaded servers, which increases robustness. We also investigate scenarios in which more than one server fails. It turns out that the best strategy to avoid overloaded servers also helps when dealing with multiple server failures. Two strategies immediately come to mind for reducing the number of overloaded servers: *(i)* virtually increasing the load of each tenant, and *(ii)* increasing the headroom left unused on each server. Experiment 5.2 inspires a third strategy: increasing the number of replicas. The intuition behind this strategy is that a higher replication factor could help smoothing out harsh load changes.

Experiment (vi): *Avoiding Overloaded Servers.* Figure 7 shows how the three over-provisioning schemes described above influence the trade-off between operating cost and the occurrence of overloaded servers. We limit ourselves to evaluate the over-provisioning strategies with robustfit-inc. For both graphs, we vary the strength of the respective strategies from left to right by increasing the headroom on the servers in steps of 0.05 and by increasing the load of the tenants by 5% in each step. Also, we increase the number of replicas per tenant by an increasing offset when going from left to right. Increasing any of these three parameters results in more active servers and the resulting placement becomes more expensive in turn. The cost of placement for our Wednesday is shown on the vertical axis. As points of reference, both graphs in Figure 7 also show the state-of-the art approaches from Experiment 5.1 as a baseline. robustfit-s-interl. (dark blue arrows) produces no temporary overloads because it strongly over-provisions. kairos-MIP (light blue arrows) produces more temporary overloads but is less expensive. Another point of reference in both charts is robustfit-inc. in the standard configuration without any over-provisioning strategy (pink arrows).

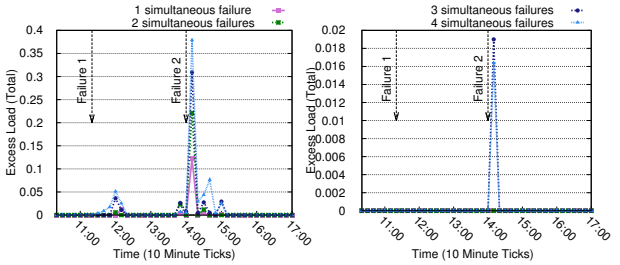
Figure 7a shows the number of occurrences of overloaded servers across all ten minute intervals. Figure 7b shows the sum of all excess load across all servers for the worst ten minute interval of our Wednesday. The latter metric is particularly sensitive. Its minimization reduces the severeness of overload situations to a negligible level. For both graphs, when moving from left to right along the x-axis, the resulting placements obviously become more and more expensive. When merely counting how often servers are overloaded across all ten minute intervals, the strategy to decrease server capacities converges towards a value of zero overloaded servers faster (and thus more inexpensively) than the strategy that virtually increases tenant load. When adding up by how much the servers are overloaded, the opposite is the case and the strategy that virtually increases tenant load converges towards zero excess load faster. However, for both metrics, the strategy to increase the replication offset is clearly superior to the other two strategies.

Experiment (vii): *Multiple Server Failures.* RTP guarantees that no server is overloaded when any one other server in the cluster fails. However, since the over-provisioning strategies introduced in the previous experiment result in placements where servers have more "headroom," we are interested in whether such placements can also handle multiple simultaneous server failures. We study two metrics, *(i)* the amount by which other servers are overloaded as a consequence of one or multiple simultaneous failures, and *(ii)* how many tenants are rendered completely unavailable when multiple servers fail at the same time. We collect the first metric, the excess load, *after* load changes have been observed, the placement algorithm has run, and all migrations have been performed. This is in contrast to Experiments 5.1 and 5.3, where excess load has been measured before the placement algorithm runs (the focus was on the *robustness* towards unanticipated load changes). Also, we consider only actual load on the servers without penalty, since we investigate failure situations in which the servers are supposed to use up the headroom allotted in the form of penalty.

We inject failures into the cluster twice during the day (marked in Figure 8 using arrows). At those points in time, we fail a fixed number of servers between zero and four. The failing servers are chosen at random. We compare the standard case where no measures for over-provisioning have been applied (Figure 8a) to an over-provisioned placement using the strategy that virtually decreases the capacity of a server (shown in Figure 8b). We parameter-

Table 4: Average number of unavailable tenants

Simultaneous failures	0	1	2	3	4
No over-provisioning	0.00	0.00	1.75	5.59	11.50
Over-Prov. Strategy:					
$\ell(t)$ scaled by factor 1.85	0.00	0.00	0.68	1.80	2.78
cap_ℓ reduced to 0.45	0.00	0.00	0.10	0.60	1.76
$r(t)$ increased by 5	0.00	0.00	0.00	0.00	0.00



(a) standard configuration (b) over-provisioned

Figure 8: Impact of multiple simultaneous failures

ized this over-provisioning strategy such that it is in the sweet-spot between the number of overloaded servers and cost. The configuration we picked is marked with a circle in Figure 7a. We observe that servers are overloaded by up to 37 % in the standard configuration with four simultaneous server failures (Figure 8a). For the over-provisioned placement (Figure 8b), a measurable impact is only visible for three and four simultaneous failures. The severity in these cases is approximately 20 times lower than without over-provisioning.

Table 4 shows the number of tenants that are temporarily rendered unavailable when injecting multiple simultaneous failures into the cluster. We vary the random seed for choosing the servers that fail and report average values. From an availability point-of-view, the replication-based strategy is the clear winner. It is also the cheapest among the three over-provisioned configurations. We conclude that although RTP guarantees that SLOs are met for only a single failure, in practice, multiple simultaneous server failures are often not problematic.

6. RELATED WORK

Previous projects have addressed variants and subproblems of the tenant placement problem, but to our knowledge none considered interleaved tenant placement to minimize the required servers, whilst guaranteeing response time SLOs and taking real-world workload traces into account.

Resource Modeling. Determining a tenant's resource requirements entails characterizing the dominant resources and quantifying how much each tenant utilizes them. For MySQL, [5] presents a technique for estimating the combined disk I/O performance in the presence of multi tenancy. The tenants' CPU and memory requirements add up linearly in their model. In [18], tenants are grouped into several SLO classes. All tenants in a class have the same response time guarantees and the same size. A server is filled with a mix of tenants from different SLO classes. A binary function determines whether the server can meet the response time requirements of its tenants. [7] introduces a logical I/O metric to characterize the dominant resource bottleneck for OLAP workloads in PostgreSQL. This I/O metric aggregates lower level metrics such as buffer pool hits and hit rates in the operating system's file system cache, which are dependent on the shared disk access behavior of concurrent queries. In [21], the focus is on in-memory databases, where disk I/O is not the dominant resource. Instead, the main resources being consumed are CPU, memory, and bandwidth between CPU and memory. This situation considerably simplifies resource modeling in contrast to the disk-based approaches. Note that besides resource modeling, both [5] and [18] also provide non-linear programs for tenant placement, but they only consider static placement and there is no notion of interleaving or load redistribution in case of failure.

Declustering Algorithms. Significant research has been devoted to declustering strategies for increasing the availability of parallel database [16, 14, 19, 24, 26]. Teradata's interleaved declustering strategy uses interleaved data placement for fast recovery, whereas chained declustering [16] and adaptive overlapped declustering [24] aim at equally redistributing work in the cluster in the case of a server failure. This redistribution is done by updating the load balancing policy when a node fails, which requires controlling the load balancing mechanism on partition granularity. All declustering strategies assume that a partition can be split further into sub-partitions and, hence, distributed across servers. This assumption does not hold in our scenario, where a tenant is considered an atomic unit and tenants are so small that there is no benefit to partitioning. Furthermore, all these strategies assume a fixed number of servers and replicas, whereas our goal is to minimize the number of active servers at each point in time. To our knowledge, only Microsoft SQL Azure [1] uses interleaved tenant placement, but does not disclose details on algorithm design or effectiveness.

Greedy Algorithms. Various greedy placement strategies have been proposed, none of them considering interleaved placement. For example [25], used as a baseline in Experiment 5.1, uses a greedy first-fit algorithm after observing the tenants' load requirements. In [20], a greedy incremental placement algorithm is proposed for adaptive distributed middleware, whereas [3] provides a greedy heuristic to automatically adjust the number of machines. Neither of these approaches considers failures and/or multiple replicas.

Notably, placement strategies for virtual machines [9, 13] share many aspects with incremental tenant placement. For instance, AutoGlobe [13] uses a trace-based approach that assesses permutations and combinations of workloads to determine a near-optimal workload placement providing specific SLOs. Similarly, [9] uses a linear program and heuristics to control VM migration. Both approaches do not consider replication to increase availability and performance.

Migration Techniques. Various systems studied protocols on how to most efficiently migrate tenants: [6] presents live migration for a decoupled storage database approach; [8] does the same in a more traditional multi-tenant setup as presented here. Our algorithms can be used with these techniques by adapting the migration overhead factors.

Optimization. Finally, the optimization community has considered the bin-packing problem for decades [11, 23]. Many variations (and the RTP is one) of the problem have been studied over the years, e.g. [2, 10, 15], however, we are not aware of approaches that take robustness towards individual server failures, as we consider it, into account. It is

the robustness (or penalty, cf. Constraint (5)) that renders existing bin-packing algorithms unusable for RTP.

7. CONCLUSION

In this paper, we introduced RTP and presented a variety of incremental data placement algorithms for multi-tenant SaaS. An evaluation with real-world data revealed that our approach leads to significant cost savings in comparison to the state of the art, while adhering to response time SLOs captured in resource models. We extended our algorithms with generic strategies for over-provisioning, so that administrators who wish to run their cluster with more headroom can, at the same time, benefit from the cost savings that come with incremental placement. Our most important findings are that *(i)* robustfit-inc. and tabu-robustfit-l. find near cost-optimal solutions in short running times, *(ii)* our over-provisioning strategies reduce the impact of load spikes to a negligible level while masking multiple simultaneous server failures from the perspective of response time SLOs; and *(iii)* the over-provisioning strategy based on increasing the replication factor is the winner among the presented approaches, from a cost, availability and cluster sizing perspective.

In future work, we will study mechanisms for dynamically adjusting the over-provisioning strategies over the day (e.g. using a varying rather than a fixed offset for the number of replicas). Given the success of machine-learning techniques in related areas, another avenue for future work is going from a re-active to a pro-active placement approach, for example via load forecasting as a preprocessing step for RTP.

8. ACKNOWLEDGMENTS

The research of Jan Schaffner was supported by SAP. The work of Franklin and Kraska was supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from SAP, Amazon Web Services, Google, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!. The authors would like to thank Marcus Krug, Rean Griffith and Marc Pfetsch for helpful discussions. The authors further thank SAP for providing the log data used for the experiments in this paper.

In memory of Dean Jacobs. Our co-author, colleague, mentor and friend Dr. Dean Bernard Jacobs passed away on January 14th, 2013 following a short severe illness. A line from one of his favorite gospel songs by Ralph Carmichael reads "There is a quiet place; far from the rapid pace; where God can soothe my troubled mind." We deeply miss Dean, his enthusiastic spirit, and his technical genius, while he rests in this quiet place.

9. REFERENCES

[1] P. A. Bernstein et al., Adapting microsoft SQL server for cloud computing. ICDE 2011

[2] C. Chekuri et al., On multi-dimensional packing problems. ACM SODA, 1999

[3] J. Chen et al., Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. IEEE ICAC 2006

[4] J. Csirik et al., Bounded Space On-Line Bin Packing: Best is Better than First. ACM SODA 1991

[5] C. Curino et al., Workload-aware database monitoring and consolidation. SIGMOD, 313–324, ACM, 2011

[6] S. Das et al., Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. PVLDB, 4(8):494, 2011

[7] J. Duggan et al., Performance prediction for concurrent database workloads. SIGMOD 2011

[8] A. J. Elmore et al., Zephyr: live migration in shared nothing databases for elastic cloud platforms. SIGMOD, 301–312, ACM, 2011

[9] T. C. Ferreto et al., Server consolidation with migration control for virtualized data centers. Future Generation Comp. Syst., 27(8):1027, 2011

[10] French Operational Research and Decision Support Society, ROADEF Challenge 2012. http://challenge.roadef.org

[11] M. R. Garey et al., Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979

[12] F. Glover, Tabu Search - Part I. INFORMS Journal on Computing, 1(3):190, 1989

[13] D. Gmach et al., An integrated approach to resource pool management: Policies, efficiency and quality metrics. DSN 2008

[14] L. Hedegard et al., The benefits of enabling fallback in the active data warehouse, Teradata. 2007

[15] D. S. Hochbaum et al., A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach. SIAM J. Comput., 17(3):539, 1988

[16] H.-I. Hsiao et al., Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. ICDE 1990

[17] J. Krüger et al., Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. PVLDB, 5(1):61

[18] W. Lang et al., Towards Multi-tenant Performance SLOs. ICDE 2012

[19] M. Mehta et al., Data Placement in Shared-Nothing Parallel Database Systems. VLDB J., 6(1):53, 1997

[20] J. M. Milán-Franco et al., Adaptive Middleware for Data Replication. ACM/IFIP/USENIX International Middleware Conference 2004

[21] J. Schaffner et al., Predicting in-memory database performance for automating cluster management tasks. ICDE 2011

[22] J. Schaffner et al., Realistic Tenant Traces for Enterprise DBaaS. SMDB, ICDE Workshops, 2013

[23] J. M. Valério de Carvalho, Exact solution of bin-packing problems using column generation and branch-and-bound. Annals of Oper. Research, 1999

[24] A. Watanabe et al., Adaptive Overlapped Declustering: A Highly Available Data-Placement Method Balancing Access Load and Space Utilization. ICDE 2005

[25] F. Yang et al., A Scalable Data Platform for a Large Number of Small Applications. CIDR, 2009

[26] H. Zhu et al., Shifted declustering: a placement-ideal layout scheme for multi-way replication storage architecture. ACM ICS 2008