A Modular Failure-Aware Resource Allocation Architecture for Cloud Computing

A.P. Chester, M. Leeke, M. Al-Ghamdi, S.A. Jarvis, and A. Jhumka

Department of Computer Science, University of Warwick, Coventry, UK, CV4 7AL
{apc,matt,mhd,arshad,saj}@dcs.warwick.ac.uk

Abstract. The prevalence of cloud computing environments and the ever increasing reliance of large organisations on computational resources has meant that service providers must operate at unprecedented scales and levels of efficiency. Dynamic resource allocation (DRA) policies have been shown to allow service providers to improve resource utilisation and operational efficiency in presence of unpredictable demands, hence maximising profitability. However, practical considerations, such as power and space, have led service providers to adopt rack based approaches to application servicing. This co-location of computation resources, and the associated common provision of utilities it encourages, has immediate implications for system dependability. Specifically, in the presence of rack crash failures which can lead to all the servers within a rack becoming unavailable, resource allocation policies need to be cognisant of failures. In this paper, we address this issue and make the following specific contributions: (i) we present a modular architecture for failure-aware resource allocation, where a performance- oriented DRA policy is composed with a failure-aware resource allocator, (ii) we propose a metric, called *Capacity Loss*, to capture the exposure of an application to a rack failure, (iii) we develop an algorithm for reducing the proposed metric across all applications in a system operating under a DRA policy, and (iv) we evaluate the effectiveness of the proposed architecture on a large-scale DRA policy in context of rack failures, ultimately concluding that our approach reduces the number of failed requests as compared to a single random allocation. The main benefit of our approach is that we have developed a failure-aware resource allocation framework that can work in tandem with any DRA policy.

1 Introduction

The emergence of cloud computing promises to the revolutionise the way that organisations manage their use of computational resources. Under traditional models, organisations must take responsibility for the procurement, installation and maintenance of costly and inflexible systems that must be periodically reconsidered. However, under a cloud-based model this is not the case. Instead organisations can consume resources as and when they are required, with cloud providers ensuring that the capacity of supplied resources can expand to meet the individual needs of any organisation.

Dynamic resource allocation (DRA) policies allow resource allocation to be matched with application workload, thus improving resource utilisation and operational efficiency in situations where workloads are volatile. The potential for DRA policies to improve utilisation and efficiency in this way is established and potentially profitable [3].

Further, the characteristics of DRA policies and the multiplicity of concerns they account for make them particularly applicable at large-scale, such as in the context of cloud computing.

In order to realise effective and economic models for cloud computing, service providers are turning to data centers and high speed networks, as well as DRA. In order to maximise resource utilisation and operational efficiency, hence profitability, a cloud provider must also make many practical considerations in the design of a data center, including those relating to physical space, connectivity, power and cooling. These considerations typically make it practical for computational resources to be arranged in collections, hence the necessity of rack based application servicing. However, this arrangement typically leaves collections of resource dependent on common utilities, such as power and network connections, as well as exposing them to a host of co-location issues, such as physical impact and disconnection.

The described environment is particularly consistent with the notion of a cloud provider offering *Infrastructure as a Service* [14], where an elastic computational resource is leased to a organisation, who must then configure the resource for use. Under this model an organisation would typically be billed for the resource utilised, this providing flexibility in situations where workloads are volatile.

Given the scale at which physical resources exist in such environments, failures will be the norm rather than the exception [6]. However, when problems such as power outage or network switches failures occur, a whole rack can fail, leading to multiple servers being unavailable. Failures at such a scale are very likely to significantly reduce the efficiency, hence profitability, of a system. Thus, two possible ways of addressing this problem of multiple serves being unresponsive are: (i) design DRA policies that are failure-aware or (ii) execute a DRA when a failure is encountered. In this paper, we adopt the first of these approaches, and reserve the second option for future work.

For the design of failure-aware DRA policies, it is unrealistic to expect organisations to modify their policies to capture failure awareness. Rather, it would be better if a failure-aware module can be designed that can work with any DRA currently in use. In this paper, we propose a modular architecture for failure-aware DRA, whereby a failure-aware allocator module works in tandem with a DRA module. The failure-aware allocator module ensures that resource allocation are made in such a way so as to minimise the impact of rack failures on running applications. The novel property of our approach is that the failure-aware allocator only performs "robustness" balancing across applications.

1.1 Contributions

In this context, we make the following specific contributions:

- We develop a modular architecture for failure-aware resource allocation.
- We propose a metric, called *CapacityLoss*, which captures the exposure of an application to a rack failure
- We develop an algorithm for reducing the *CapacityLoss* for each application in a system, given the resource requirements of each application.

– We evaluate the effectiveness of our framework on a large-scale DRA policy in context of rack failures, and show the efficiency of our approach.

The overarching contribution of this paper is the development of a modular failure-aware architecture for resource allocation. The novel aspect of this is that the failure-aware resource allocator can work well in tandem with any DRA policy that is being used by the organisation.

1.2 Paper Structure

The remainder of this paper is structured as follows: In Section 2 we provide a survey of related work. In Section 3 we set out the system and fault models adopted in this paper. In Section 4 we define and illustrate a machine placement metric for improving the dependability of existing DRA policies. In Section 5 we detail our experimental setup, before presenting the results of our experimentation in Section 6. In Section 7 we discuss the implications of the results presented. Finally, in Section 8 we conclude this paper with a summary and a discussion of future work.

2 Related Work

The properties of DRA policies have been throughly considered in the context of performance evaluation and measurement. In this context, the use of dynamic resource allocation has been shown to improve application performance and the utilisation of resources [3] [10]. Other work relating to the performance of DRA policies had led to the development, augmentation and evaluation of policies with a view to further enhancing DRA system performance [1] [17]. For example, in [4] the authors developed an algorithm for profit maximisation of multi-class requests through dynamic resource allocation, whilst work in [2] proposed an approach for the dynamic allocation of resources from a common pool. The issue of resource allocation is also addressed in research such as [12], where periodic balanced server provisioning was investigated. In contrast to experimental research, analytical approaches to dynamic resource allocation have also been explored in work such as [7], [13] and [18], where application modelling and queuing theory were applied respectively.

Resource failures in the context of data centers is well addressed by existing literature[8] [15] [16]. The coverage of resource failures has mostly concerned failures affecting a single hardware resource, excluding the possibility of large scale failures. The authors of [15] give an insight into the failure rates of servers in a large datacenter, and attempt to classify them using a range of criteria. Work in [6] develops a cloud ready platform for testing a range of failure scenarios, including rack based failures. This demonstrates the need for systems which are able to mitigate against large-scale failures. The issue of resource failures in cloud computing is addressed in [11], where the authors develop a policy to partition a resource between high-performance computing application and web service applications in the context of single resource failures. The work in this paper increases the scale of the resource failure and in doing so increases the number of applications affected by the failure.

The issues of rack-awareness has been considered to some extent by [9], part of which is a file system, known as HDFS, that can account for rack distribution when storing data. Our work differs from [9] as our algorithm works in conjunction with a DRA system at the level of Infrastructure as a Service while Hadoop operates at the level of *Platform as a Service*.

3 Models

In this section, we present the system and fault models adopted in this paper as well as enunciating our assumptions.

3.1 System Model

We consider an environment where a set of applications $A = \{a_1, a_2, ..., a_n\}$ are deployed across a set of racks $R = \{r_1, r_2, ..., r_m\}$. A rack r_i consists of a set of servers $S_i = \{s_{i,1}, s_{i,2}, ..., s_{i,k}\}$. A server $s_{i,j}$ may service requests from only one application at any time. We assume that (i) all servers $s_{i,j}$ are homogeneous in that they provide identical resources, and (ii) each server has all the resources required by an application, i.e., no communication between servers is required for application servicing. Such a system model is typical in cloud computing environment or large scale datacenters.

3.2 Fault Model

Given the described system model, failures are expected to be the norm rather than the exception [6] due to the very large number of computing hardware present. We consider crash failures to occur at a rack level, where this type of failure may be caused due to, for example, power outages or network switch failures. When such a rack r_i fails, all the servers $s_{i,j} \in S_i$ become unavailable. We assume a failure mode where at most a single rack can fail at any one time.

4 A Failure-Aware Allocation Algorithm

4.1 Metric for Failure-Awareness

Well designed enterprise systems can allow for increased throughput via horizontal scaling of resources with minimal overhead [5]. The system may scale linearly with the addition of resources which do not overload any of the other tiers. In a homogeneous server environment of n servers, each server contributes $\frac{1}{n}$ of the total system capacity.

A rack failure will impact on multiple applications depending on the composition of the rack which is affected. To meet performance requirements, resource allocation will have to take place. However, the resource allocation needs to be cognisant of the failures. For example, in the Hadoop distributed file systems [9], an application will be located on at least two different racks, so that failure of one rack does not cause the application to become unavailable. To assess the impact of the failure of rack $r_i \in R$ on a given application $a_j \in A$, we use the proportion of servers hosting application a_j that

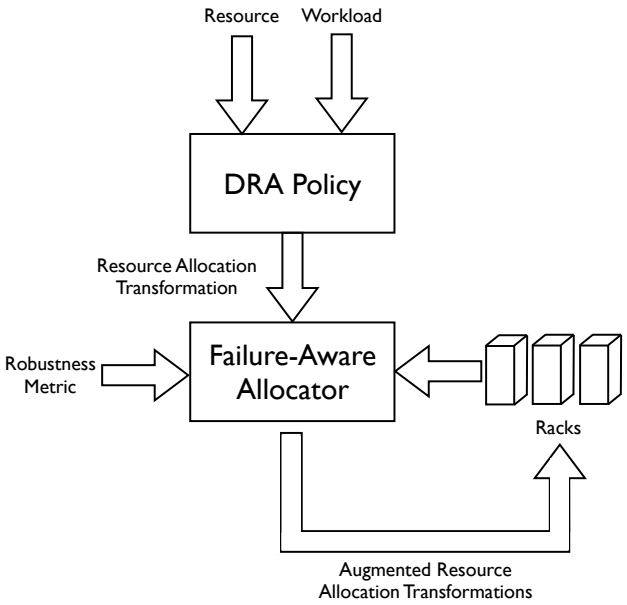

Fig. 1. A Modular Architecture for Failure-Aware Resource Allocation

will be lost due to the failure of the rack r_i to measure the loss of application capacity for a_j, and base reallocation decisions on that proportion.

The metric we propose, called *capacity loss*, is shown as equation 1, where r_i is the rack, a_j is an application and s_i^j is the number of servers in rack r_i hosting application a_j, and S_{a_j} is the number of servers hosting application across all racks, is used to capture the impact the failure of rack r_i will have on the application a_j in terms of capacity.

$$CapacityLoss_{r_i,a_j} = \frac{s_i^j}{S_{a_j}} \tag{1}$$

It is clear for Equation 1 that a *Capacity Loss* of 1 is undesirable, as it would means that all servers hosting application a_j are located in rack r_i. In an ideal situation, the *Capacity Loss* value would be 0. Thus, an objective is to minimise *Capacity Loss* for all applications across all racks.

4.2 A Modular Architecture for Failure-Aware Resource Allocation

In this section, we present a modular architecture, in Figure 1, for failure-aware resource allocation in the presence of rack failures.

In Figure 1, the DRA component represents the dynamic resource allocation algorithm that is used by the cloud computing environment. The DRA component takes the forecasted application workload as input, and outputs a set of resource allocation transfor-

mation for performance reasons. For example, such an output will be of the following type:

- Take 4 servers from application 4 to host application 2.
- Take 2 servers from application 1 to host application 3.

However, such DRAs are not rack-aware. Specifically, DRA outputs do not specify the racks where the resource transformation is to take place. In this paper, one of our contributions is to develop a component, namely the *Allocator* (see Figure 1), that performs the rack-aware resource transformation in that it augments the resource transformation needed with rack information, i.e., it specified the racks where the transformations are to occur. For example, with an allocator module, our previous example would be like:

- Take 4 servers from application 4 to host application 2 (2 servers from rack 1 and 2 servers from rack 3).
- Take 2 servers from application 1 to host application 3 (1 server from rack 2 and 1 server from rack 2).

However, with the non-zero probability of rack failures, the allocator needs to be failure-aware in that it needs to perform these resource transformation in such a way to minimise the *Capacity Loss* (see Equation 1) for each application. The algorithm for the failure-aware allocator is shown in Algorithm 1. In a nutshell, Algorithm 1 searches for all racks where application a_f is being hosted. Then, the failure-aware allocator chooses the a set of racks, denoted R_f^t, where the capacity loss for application a_f on a rack r_i, where $r_i \in R_f^t$ is maximum, and capacity loss for application a_t on r_i is minimum. Practically, this implies that it is better to remove servers for application a_f from a rack $r_i$where the capacity loss for a_f is already high. Further, it also means that it is better to allocate servers to host application a_t on rack r_i where application a_t already has minimum capacity loss. Once these racks R_f^t are identified, one of them is picked at random.

In the next sections, we describe and present the results of experiments designed to evaluated the efficiency of our architecture and failure-aware allocator algorithm.

5 Experimental Setup

In this section we detail the experimental approach used to derive the results presented in Section 6.

5.1 Failure Scenario

In our experimentation, we considered a DRA policy that executes periodically. Figure 2 describes a typical series of events in our experimental environment. At t_i, the DRA policy makes a decision based on the application workloads and returns a set of resource transformations (see example in Section 4) to meet the performance requirements. This set of transformations is taken as the input to the failure-aware allocator component, which then decides on which racks to enable these transformations. Specifically, the

Algorithm 1 Balanced migration algorithm

Input: a_t The application to be migrated to
Input: a_f The application to be migrated from
Input: R the set of all racks
Output: The optimal rack for migration

Let $c_1 = \emptyset$
Let $c_2 = \emptyset$
Let $c_3 = \emptyset$
Let o be an optimal rack
Let m_t represent the best value for a_t
Let m_f represent the best value for a_f
Initialise $m_t = +\infty$
Initialise $m_f = -\infty$
for all $r \in R$ **do**
    if r contains a_f **then**
        $c_1 \leftarrow r$
    end if
end for
for all $r \in c_1$ **do**
    $v_t = CapacityLoss_{r,a_t}$
    if $v_t \leq m_t$ **then**
        if $v_t < m_t$ **then**
            $c_2 = \emptyset$
        end if
        $c_2 \leftarrow r$
        $m_t = v_t$
    end if
end for
for all $r \in c_2$ **do**
    $v_f = CapacityLoss_{r,a_f}$
    if $v_f \geq m_f$ **then**
        if $v_f > m_f$ **then**
            $c_3 = \emptyset$
        end if
        $c_3 \leftarrow r$
        $m_f = v_f$
    end if
end for
return Random member of c_3

failure-aware resource allocator module augments the resource transformations with rack location information. A rack failure t_f occurs such that $t_i < t_f < t_{i+1}$, where t_{i+1} is the period for the next DRA decision.

Thus, with the proposed modular failure-aware DRA framework, we expect the system to better handle rack failures. To do this, we measure the number of failed requests between t_f and t_{i+1}. It is the aim of the research presented in this paper to minimise

the system performance loss incurred in the period between t_f and t_{i+1} through rack-awareness in the resource allocation mechanism.

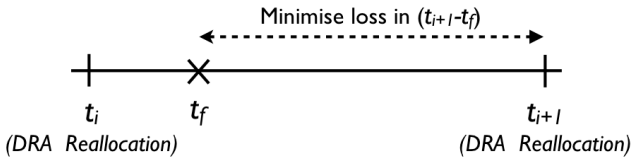

Fig. 2. Timeline depicting a typical event ordering

5.2 Resource Allocation

Much current research in DRA policies does not explicitly consider resource failure. In such works, each application is viewed as a logical collection of servers which are equal. In this case, servers may be migrated between applications arbitrarily. We use this naïve approach as our benchmark, selecting servers to be migrated as required with no regard for their location. We refer to this allocation mechanism as the *random allocator*.

Under the balanced resource allocator, which uses the *Capacity Loss* metric shown in Equation 1 to assign applications, we attempted to minimise the potential capacity loss in the event of a rack failure. In Section 4, we decoupled the failure-aware allocation mechanism, in the allocator, from an abstract DRA policy. For our experiments we employed this approach and used the resource allocation algorithm shown in Algorithm 2. The algorithm uses workload prediction and applications which are ranked by criticality (this may be governed by SLA or a business metric) to partition resources.

To reduce experimental uncertainty we used 10 identical applications. Each application processes a single type of request with a fixed service duration of 100ms.

The simulated datacenter is comprised of 400 homogeneous servers that are housed in racks. Each rack contained 40 servers, giving a total of 10 racks. Initially the servers were allocated evenly between applications, i.e., 40 servers per application, with 4 servers per application per rack. The initial allocation of servers to racks was done such that the minimum capacity loss for each application, i.e., maximum robustness for each application, was achieved. When a server was reallocated, it first had to complete the servicing of current and queued requests, before migrating to the newly assigned application. The process of server migration was fixed at 30 seconds.

As the number of applications is high, we use synthetic sine-based workloads of various frequencies and amplitudes. In all cases the total workload of the system is greater than 75% of total system capacity. The individual workloads are shown in Figure 3.

In this paper, we use a predictive resource allocation algorithm that uses an exponential moving average to forecast the workload for the next interval and allocates resources accordingly. The algorithm ranks applications in accordance with their importance (this may be governed by SLAs or a business metric) and greedily allocates re-

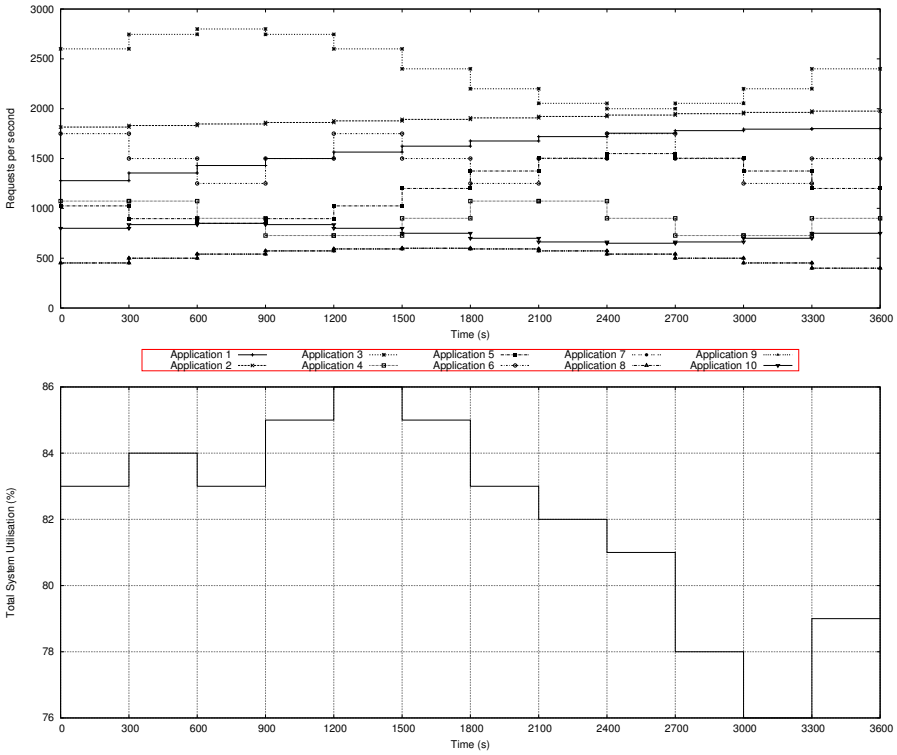

Fig. 3. Experiment workloads

sources to applications in order of priority. This algorithm allocates all server resources to applications. The complete allocation algorithm can be seen in Algorithm 2.

We selected five timings for rack failures. Each rack was then failed at each of the timings in a separate simulation. The times selected for the failures are 645, 1245,1845, 2445, 3045 seconds. The total simulation time was 3600 seconds.

5.3 Expected Results

Based on our proposed framework, we expect the following result from our experiments, which we will verify in the results section (Section 6):

- We generally expect the balanced resource allocator (Algorithm 1) to have less failed requests due to the fact that it creates allocation which minimises the capacity loss for each application across each rack.
- We anticipate that the random allocator may yield better performance in rare situations, where a rack is lightly leaded with respect to a single application, though this is expected to be offset by a higher application exposure, i.e, a high value of $CapacityLoss$, for an application on the same rack.

Algorithm 2 Improved scalability algorithm

Let N be the number of applications
Let $a_1, ..., a_n$ be ranked applications
Let $m_1, ..., m_n$ be minimum resource of a_i
Let t_a be the throughout of a server of a
Let n_a be number of servers allocated to a
Let I be idle servers
for $i = 1; i < N; i++$ **do**
    p = predictedDemand(a_i)
    S'_i = $\frac{p}{t_a}$
    if $S'_i < S_i$ **then**
        Append $S_i - S'_i$ servers to I'
        $S'_i \rightarrow S_i$
    end if
end for
for $i = 1; i < N; i++$ **do**
    if $S_i \neq S'_i$ **then**
        if $I' > 0$ **then**
            if $I' > S'_i - S_i$ **then**
                Move $S'_i - S_i$ from I' to S_i
                break
            else
                Move all servers in I' to S_i
            end if
        end if
        for $j = N; j > i + 1; j--$ **do**
            if $S'_i - S_i \leq S_j - m_j$ **then**
                Move $(S'_i - S_i)$ from S_j to S_i
                Break
            else
                Move $(S_j - m_j)$ from S_j to S_i
            end if
        end for
    end if
end for
Let $I = I'$

6 Results

In this section, we present the results of our experimentation. The results presented give (i) details of the overall system impact, (ii) analysis for each failure timing and (iii) discussion of the effect of failures on each application. Any difference is between the balanced and the random allocator, where a positive difference represents better performance for the balanced allocator.

Table 1. Maximum total request failure percentage for each failure

Allocator	Failure 1	Failure 2	Failure 3	Failure 4	Failure 5
Random	3.20	2.83	2.42	1.87	1.60
Balanced	3.16	2.75	2.27	1.85	1.60
% Imp.	1.12	2.87	5.95	1.09	0.02

Table 2. Standard deviation of failures across all racks

Allocator	Failure 1	Failure 2	Failure 3	Failure 4	Failure 5
Random	0.0588	0.0373	0.0710	0.0433	0.0105
Balanced	0.0475	0.0309	0.0358	0.0237	0.0096
% Imp.	19.1851	17.1305	49.4922	45.2765	7.9847

6.1 Overall Results

We now present the overall impact of a rack failure on the total failure rate over the duration of the experiment. Firstly, we consider the impact of the allocation technique on the maximum percentage of failed requests for each failure. Each value shown in Table 1 reflects the percentage of failed requests over the duration of a simulation. The results demonstrate that the balanced allocator reduced the maximum impact of a rack failure on the overall failure rate, as compared with the random allocator, as it consistency yields an improvements in the percentage of failed requests. The maximum observed improvement is nearly 6%. This is a considerable improvement, given that it was measured over the full duration of a simulation.

The standard deviation of the overall failure percentage for each rack and failure is shown in Table 2. The nature of the balanced allocator caused a consistent reduction in the standard deviation of the failure rates. This can be explained by the fact that the balanced allocator algorithm (Algorithm 1) attempts to reduce each application's *Capacity Loss*, thereby reducing the deviation in terms of failed requests. The most significant improvement can be seen in Failure3, though it should be noted that the lowest percentage improvement is nearly 8%.

6.2 Failure Results

Failure 1 in our experimentation occurs at 645 seconds, after two migration intervals. Table 3 contains the failures observed for each application, under both the random and balanced allocators. As mentioned before, we crashed all 10 racks at the 645 seconds marks. The minimum/maximum columns represent the minimum/maximum number of failed requests across all the 10 failures. The rightmost column of Table 3 presents the percentage improvement in standard deviation of the balanced allocator over the random allocator.

We make the following two observations: For all applications, (i) the minimum number of failed requests under the random allocator is equal to or better than the balanced allocator, and (ii) in terms of maximum values, the balanced allocator performs better than the random allocator. The first observation is due to the fact that the random

Table 3. Application results for Failure 1

	Random				Balanced				
App.	Min	Max	Average	Std.Dev.	Min	Max	Average	Std.Dev.	Improvement (%)
1	1995	2494	2145	240.83	1995	2494	2145	240.83	0.00
2	2015	3526	2770.5	593.26	2518	3023	2770.5	265.53	55.24
3	2244	43213	19794	15027.35	12248	22572	17410	5440.17	63.80
4	794	1589	1350	277.62	1191	1589	1350	205.05	26.14
5	1034	1379	1275	166.08	1033	1379	1275	166.77	-0.42
6	1223	2446	1875	438.36	1630	2039	1875	210.65	51.95
7	507	14749	2083.5	4456.62	507	1015	811.5	262.07	94.12
8	531	21216	4987	8563.51	1062	1594	1275	274.12	96.80
9	78896	120352	95478.4	13990.33	89260	99624	95478.4	5351.95	61.75
10	0	20332	6293.6	8381.72	564	10008	4341.6	4876.86	41.82

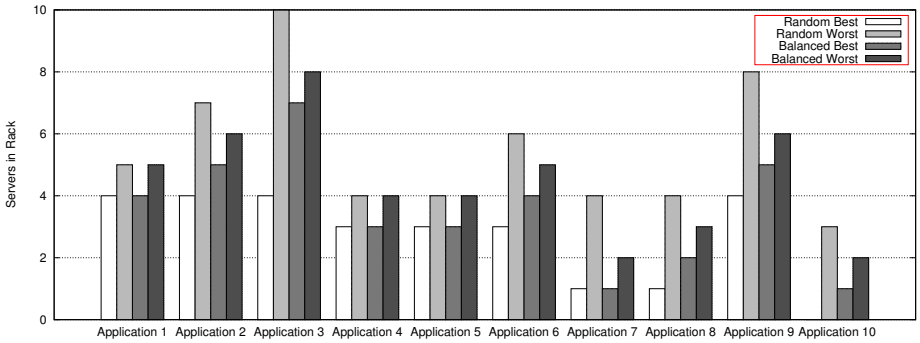

Fig. 4. Range of allocations giving best and worst results for Failure1

allocator can place fewer instances of an application on a rack, i.e., an application can have a capacity of 0 on a rack, which never happens in the balanced algorithm (unless an application has a 0 workload), than the balanced allocator. On the other hand, for the maximum values, the random allocator may allocate servers in a rack to applications in such a way that the *Capacity Loss* is very high, resulting in very high number of failed requests.

Table 3 shows the difference in standard deviation between the two allocators. The result for Application 5 is due to a minor difference in the round robin scheduling of requests across servers. Application 1 has an identical failure rate under both allocators. The best and worst rack configurations for each application under both allocators are shown in Figure 4. The maximum difference in allocations for the random policy is 6, while the balanced policy achieves a maximum difference of 1, which shows that Algorithm 1 of the balanced allocator achieves better "robustness" balancing than the random allocator.

This pattern of results is corroborated by the results from failures 2-5, which are given in Tables 4-7. This supports the first of our expected results, as proposed at the end of Section 5

Table 4. Application results for Failure 2

	Random				Balanced				
App.	Min	Max	Average	Std.Dev.	Min	Max	Average	Std.Dev.	Improvement (%)
1	2086	2609	2347.5	275.22	2086	2609	2347.5	275.22	0.00
2	2012	4025	2817	679.31	2515	3018	2817	259.49	61.80
3	2026	5062	3900	894.12	3545	4053	3900	244.74	72.63
4	703	1407	1090.5	259.53	1055	1408	1090.5	111.56	57.02
5	904	1809	1537.5	316.39	1356	1809	1537.5	233.24	26.28
6	22848	64304	46685.2	10979.10	43576	53940	46685.2	5006.29	54.40
7	523	17533	2433.5	5311.79	523	1047	889.5	252.91	95.24
8	0	9060	1856	2590.29	1000	1500	1200	258.20	90.03
9	41822	83278	61513.4	11405.61	52186	62550	61513.4	3277.31	71.27
10	0	1046	732	441.01	522	1046	732	270.25	38.72

Table 5. Application results for Failure 3

	Random				Balanced				
App.	Min	Max	Average	Std.Dev.	Min	Max	Average	Std.Dev.	Improvement (%)
1	2095	3143	2514	331.34	2095	2619	2514	220.83	33.35
2	2044	4088	2862	689.52	2555	3067	2862	264.01	61.71
3	1784	4459	3300	703.31	2675	3568	3300	311.94	55.65
4	1073	5906	2361.5	1880.91	1073	5907	1985.5	1388.15	26.20
5	31948	52676	45421.2	8532.40	42312	52676	45421.2	5006.29	41.33
6	852	2556	1875	500.00	1704	2131	1875	219.90	56.02
7	494	17911	2433.5	5443.78	494	989	889.5	208.45	96.17
8	437	1751	1050	422.90	874	1313	1050	226.35	46.48
9	2064	3615	3045	513.64	2580	3098	3045	163.39	68.19
10	0	1326	619.5	427.38	442	885	619.5	228.51	46.53

6.3 Application Results

Table 8 gives the range of improvement provided by the balanced allocator across all failures and across all racks. The benefit of the balanced allocator varies between -0.42% and 96.80% across each application.

In two cases Application 1 gains no benefit form the allocator, due to identical allocations from both policies. The balanced allocator improves the deviation of Application 1 by an average of 29.27%.

When using the balanced allocator Application 5 has a marginally worse variance (-0.42% and -0.25%) than the random allocator. This is due to the round-robin scheduling of requests to servers causing a slightly higher load at the point of failure.

7 Discussion

The results presented have demonstrated the need for failure awareness to be incorporated into system which operate under DRA policies. The most desirable approach for achieving this would be one that requires no modification to the DRA policies already

Table 6. Application results for Failure 4

	Random				Balanced				
App.	Min	Max	Average	Std.Dev.	Min	Max	Average	Std.Dev.	Improvement (%)
1	2064	3611	2631	451.50	2579	3095	2631	163.03	63.89
2	2037	4075	2904	681.62	2547	3058	2904	246.12	63.89
3	1688	4226	3000	703.48	2535	3381	3000	240.17	65.86
4	843	2110	1350	387.89	1265	1688	1350	177.88	54.14
5	32968	64060	51623.2	8175.23	43332	53696	51623.2	4369.85	46.55
6	11282	52721	36145.6	12160.21	32001	42362	36145.6	5349.79	56.01
7	427	2135	811.5	549.43	427	855	811.5	135.10	75.41
8	443	1773	975	407.36	886	1329	975	186.57	54.20
9	1308	3055	2355	512.30	2180	2617	2355	225.28	56.03
10	0	3567	778.5	1028.41	468	938	562.5	197.64	80.78

Table 7. Application results for Failure 5

	Random				Balanced				
App.	Min	Max	Average	Std.Dev.	Min	Max	Average	Std.Dev.	Improvement (%)
1	2032	3556	2692.5	481.90	2540	3048	2692.5	245.32	49.09
2	2030	4063	2944.5	710.41	2537	3047	2944.5	214.51	69.80
3	1860	4647	3300	772.55	2788	3718	3300	263.86	65.85
4	681	1704	1090.5	313.16	1022	1364	1090.5	144.15	53.97
5	1964	2455	2062.5	206.87	1964	2456	2062.5	207.39	-0.25
6	852	2557	1875	500.19	1704	2131	1875	220.12	55.99
7	357	1784	678	458.89	357	714	678	112.79	75.42
8	500	1500	1050	437.80	1000	1500	1050	158.11	63.88
9	1168	2337	1869	402.15	1557	1947	1869	164.44	59.11
10	0	2553	1075.5	1031.49	516	2554	923.5	858.82	16.74

in use, i.e., the most desirable approach would be composing the DRA policy itself with a failure-aware allocator component. The modular architecture proposed in this paper has shown that, by decoupling the allocation mechanism from for DRA policy, the performance-oriented goals of the DRA can be separated from the conflicting aim of improving application robustness.

Once the decoupling of the DRA policy and allocation mechanism has been achieved, the focus turns to the function of the allocation mechanism itself. The metric proposed in this paper, $capacityLoss$, serves to quantify the exposure of an application, e.g., where $capacityLoss = 1$ for a given application, that application is completely exposed to a failure of the rack on which it is hosted. This focus on application exposure is motivated by the fact that, while it is reasonable to balance applications across racks in the context of static allocation, this is not possible under existing DRA policies, which assume the existence of a common, un-partitioned resource pool.

The proposed modular architecture and balanced allocation mechanism have been shown to reduce the impact of failures on applications being serviced across racks. Interestingly, the random allocation mechanism was shown to provide the lowest absolute

Table 8. Application Results Across Failures and Racks (% Improvment In Standard Deviation)

App.	Min (%)	Max (%)	Average (%)
1	0%	63.89%	29.27
2	55.24	69.80	62.49
3	55.65	72.63	64.76
4	26.14	57.02	43.49
5	-0.42	46.55	22.70
6	51.95	56.02	54.87
7	75.41	96.17	87.27
8	46.48	96.80	70.28
9	56.03	71.27	63.27
10	16.74	80.78	44.92

impact for any single application; when the failed rack was lightly loaded with respect to the application, i.e., when the capacity loss of that application on a rack was minimal. However, this is not necessarily positive, as for this light loading to occur, another rack may be heavily loaded. Results averaged across all racks demonstrate significant improvements in the mean number of failed requests for the balanced allocator. In addition to the average case, the balanced allocator exhibited lower deviations than the random allocator. Since rack failures are inherently unpredictable, minimising the average case is clearly of great benefit when operating at large-scale. In turn, this represents lesser exposure to loss of income due to unforeseen unavailability of applications.

As the magnitude of the systems operating under DRA policies increases, the scale and frequency of the problems addressed in this paper will similarly increase. Hence, as failures become the expectation rather than the exception for large-scale systems, effective resource allocation and modular architectures that facilitate a separation of concerns will become increasingly important for cloud providers offering Infrastructure as a Service.

8 Conclusion and Further Work

In this section, we provide a contribution summary and discussion of future work relating to the results presented.

8.1 Summary

In this paper, we have made the following novel contributions: (i) we have presented a modular architecture for failure-aware resource allocation, where a performance-oriented DRA policy is composed with a failure-aware resource allocator, (ii) we have proposed a metric, called *CapacityLoss*, to capture the exposure of an application to a rack failure, (iii) developed an algorithm for reducing the proposed metric across all applications in a system, and (iv) evaluated the effectiveness of the proposed architecture on a large-scale DRA policy in context of rack failures in order to show the efficiency of our approach.

The results presented demonstrate the effectiveness of our architecture and mechanisms, with consistent improvement being observed in almost all situations. Indeed, when the average case is considered, the proposed approach exhibits a minimum improvement of more than 22% and a maximum of more than 87% across all failure situations.

The novelty of our modular, failure-aware architecture for resource allocation is that it is applicable to, and will work in tandem with, any DRA policy. Practically, this implies that organisations need not modify their DRA policy, rather they can just integrate the failure-aware resource allocator to reduce the exposure of applications to rack failures.

8.2 Future Work

As virtualisation technologies continue to improve and service providers look to consolidate their servers, it is inevitable that application densities will increase. In future work we intend to consider the effects that such consolidation will have on the issues explored in this paper. For example, as application densities and further complexity is incorporated in to the software of resources, it may be necessary to reconsider how application exposure can be measured.

Additionally, in this paper we have demonstrated the ability of a modular architecture and a balanced allocator to reduce the average number of failed requests in a large-scale system. This exploration could be expanded through the development of a, still decoupled, allocation mechanism that can account for rack failures in an active manner, thus reducing the need to reallocate.

References

1. M. Al-Ghamdi, A. Chester, and S. Jarvis. Predictive and dynamic resource application for enterprise applications. In *Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications*, pages 2776–2783, June 2010.
2. J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing*, pages 90–100, June 2003.
3. A. P. Chester, J. W. J. Xue, L. He, and S. A. Jarvis. A system for dynamic server allocation in application server clusters. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, pages 130–139, December 2008.
4. L. He, W. J. Xue, and S. A. Jarvis. Partition-based profit optimisation for multi-class requests in clusters of servers. In *Proceedings of the International Conference on e-Business Engineering*, pages 131–138, October 2007.
5. J. Idziorek. Discrete event simulation model for analysis of horizontal scaling in the cloud computing model. In *Proceedings of the Winter Simulation Conference*, pages 3004–3014, December 2010.
6. P. Joshi, H. S. Gunawi, and K. Sen. Prefail: Programmable and efficient failure testing framework. Ucb/eecs-2011-3, University of California at Berkeley, 2011.
7. X. Liu, J. Heo, and L.Sha. Modeling 3-tiered web applications. In *Proc. International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, 2005.

8. D. A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX Systems Administration Conference*, pages 185–188, November 2002.
9. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10, May 2010.
10. J. Slegers, I. Mitriani, and N. Thomas. Evaluating the optimal server allocation policy for clusters with on/off sources. *Journal of Performance Evaluation*, 66(8):453–467, August 2009.
11. G. Tian and D. Meng. Failure rules based node resource provisioning policy for cloud computing. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, pages 397–404, December 2010.
12. B. Urgaonkar and A. Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 217–228, September 2005.
13. B. Urgaonkar, G. Pacifi, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems*, June 2005.
14. L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, January 2008.
15. K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 193–204, June 2010.
16. M. Wiboonrat. An empirical study on data center system failure diagnosis. In *Proceedings of the 3rd International Conference on Internet Monitoring and Protection*, pages 103–108, June 2008.
17. J. Xue, A. Chester, L. He, and S. Jarvis. Dynamic resource allocation in enterprise systems. In *Proceedings of the 14th International Conference on Parallel and Distributed Systems*, pages 203–212, December 2008.
18. Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytical model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th International Conference on Autonomic Computing*, June 2007.