Performance and Availability Aware Regeneration For Cloud Based Multitier Applications

Gueyoung Jung† Kaustubh R. Joshi‡ Matti A. Hiltunen‡ Richard D. Schlichting‡ Calton Pu†

†College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{gueyoung.jung,calton}@cc.gatech.edu

‡AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{kaustubh,hiltunen,rick}@research.att.com

Abstract

Virtual machine technology enables agile system deployments in which software components can be cheaply moved, replicated, and allocated hardware resources in a controlled fashion. This paper examines how these facilities can be used to provide enhanced solutions to the classic problem of ensuring high availability while maintaining performance. By regenerating software components to restore the redundancy of a system whenever failures occur, we achieve improved availability compared to a system with a fixed redundancy level. Moreover, by smartly controlling component placement and resource allocation using information about application control flow and performance predictions from queuing models, we ensure that the resulting performance degradation is minimized. We consider an environment in which a collection of multitier enterprise applications operates across multiple hosts, racks, clusters, and data centers to maximize failure independence. Simulation results show that our proposed approach provides better availability and significantly lower degradation of system response times compared to traditional approaches.

1. Introduction

High availability and low response time are crucial, although often conflicting, requirements for the multitier applications that implement critical business functionality for many enterprises. Ensuring high availability requires the applications to be deployed with sufficient redundancy, potentially spanning several data centers, while distributed deployment and replication impose a performance penalty. Redundancy is traditionally ensured by using reliable hardware components with high mean time between failures (MTBF) and quick repair or replacement of failed components, i.e., low mean time to repair (MTTR). However, current trends in system and data center design are changing the role of repair. Multitier systems are increasingly running on large numbers of cheap, less reliable commodity components, thus leading to a decrease in MTBF of the system components. For example, Google reported an average of 1000 node failures/yr in their typical 1800 node cluster for a cluster MTBF of 8.76 hours [11]. Meanwhile, skilled manpower is quickly becoming the most expensive resource, thus encouraging data center operators to achieve economies of scale by batching repairs and replacement, and increasing MTTR in the process. In fact, portable "data-center in a box" designs (e.g., [15]) that contain tightly packed individual components that are completely non-serviceable, i.e., with an infinite MTTR, are emerging.

These trends imply that applications will increasingly operate in environments in which parts of the infrastructure are in a failed state. Replication of software components is a standard technique used to ensure high availability. The level of redundancy must be high enough to tolerate additional failures until repairs eventually take place. Maintaining such redundancy under the low MTBF and high MTTR conditions is expensive (e.g., cost of hardware and software licenses) and may have a significant performance overhead (e.g., state replication). Meanwhile, reducing effective time-to-repair by maintaining standby spare resources that can be quickly deployed automatically is inefficient because the spares represent unutilized resources.

We present a solution that ensures high availability while maintaining good performance by employing all system resources (e.g., no idle standby resources) and limited levels of replication. Specifically, when a hardware resource fails, we regenerate the affected software components and deploy them on the remaining resources so that the required availability and performance goals of all the applications in the system are met as long as possible. The regeneration-based approach can provide high availability

with far less redundancy than static replacement oriented designs.

The proper placement, and resource allocation, for the regenerated components is crucial for ensuring high availability and low response time of the applications sharing the computing resources. The placement algorithm has to deal with the conflicting goals of performance and availability. For high availability, replicas of a component may need to be placed in different clusters or even different data centers, while the resulting increased network latency may affect the application's response time. Ensuring good performance becomes particularly challenging with multitier applications, where poor placement and resource allocation of a component (e.g., database server) may cause it to be the bottleneck for the whole application and as a result, the hosts where the other tiers of the application are placed may become underutilized. When multiple applications share hardware resources (e.g., in cloud computing), the problem becomes even more complex.

We consider a set of multitier applications and a shared pool of resources distributed across multiple racks, clusters, and data centers for greater failure independence. We present a management framework that reconfigures this set of applications on the resource pool in reaction to failures to maintain a user-defined availability level for as long as possible, and to do so in a way that minimizes response time degradation. To preserve performance, our solution not only regenerates and places the failed components but may also redeploy those components that were not impacted by the failure. We introduce a novel hierarchical optimization algorithm that balances the needs of availability and performance to produce target configurations that exhibit the least amount of performance degradation.

2. Architecture

We consider a consolidated data-center environment in which a set of multitier applications A are to be deployed across a set of physical hosts H located across a number of data centers. The physical hosts are organized into hierarchical groupings of racks, clusters, and data centers to facilitate networking and management. These groupings are represented by a resource hierarchy $(R, \leq_R)$, where R is the set of "resource tiers" (i.e., machine, rack, cluster, datacenter) and $\leq_R$ specifies hosting relationship between these tiers, e.g., Host1 $\leq_R$ Rack1 $\leq_R$ DataCenter1. Figure 1 shows an example resource hierarchy with 20 machines distributed across four racks in two data centers. Two resource tiers are said to be at the same "level" $rl \in RL$ if they are of the same type, e.g., Rack1 and Rack2. The example in the figure has three levels. The relation $\leq_R^*$ represents the transitive closure of the hosting relation $\leq_R$ such that Host1 $\leq_R^*$ DataCenter1 indicates that Host1 is hosted in DataCenter1 either directly or indirectly.

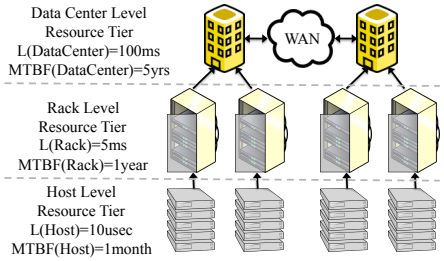


Figure 1. Resource Levels Example

Hosts are interconnected by a data center network and the network latency between hosts depends on how close they are to one another in the hierarchy, i.e., hosts placed in the same rack have a lower network latency between them than hosts across different racks, which have a lower latency than hosts in different data centers. We denote by $L(rl)$ the maximum latency between two hosts separated at resource level rl. Finally, we denote the mean time between failures for each resource tier r by MTBF_r. In general, MTBF increases with increasing resource level, i.e., MTBF for hosts is smaller than MTBF for a rack, which is smaller still than the MTBF for an entire data center.

Each application a consists of a set N_a of component types (e.g., web server, database), each of which contains several replicated components. The number of replicas of type $n \in N_a$ for application a are given by $\mathrm{reps}(n)$. To avoid single points of failure, the replication level for each component type must be at least 2. Each application a may support multiple transaction types T_a. For example, the RUBiS [5] auction site benchmark used in our testbed has transactions that correspond to login, profile, browsing, searching, buying, and selling. Each transaction can initiate a sequence of function calls between application components. The workload w_a for the application can be characterized by the set of request rates for each of its transactions, i.e., $\{w_a^t | t \in T_a\}$, and the workload for the entire system as $W = \{w_a | a \in A\}$. Each application component replica n_k executes in its own Xen virtual machine [3] on a physical host that it can share with other VMs. Each VM is allocated a fractional share of the host's CPU denoted by $\mathrm{cap}(n_k)$ that is enforced by Xen's credit-based scheduler. VMs belonging to a single application may be hosted on any physical host anywhere in the resource hierarchy. Generally, the higher the resource tier across which an application's VMs are distributed, the worse is its performance due to higher network latency.

Finally, as is common in data center environments, we assume a shared storage network (e.g., a SAN), so that any VM residing on a server that has failed can be reinstantiated on another host by using its disk image. For resource tiers across which a SAN is not available (e.g., across data cen-

ters), VM reinstantiation can still be achieved by copying the VM's disk image to the target, and subsequently executing the VM. Although our approach could also be applied to disk failures, in this paper we assume they are handled within the storage array using RAID technology. If protection against loss of volatile data is needed, it is assumed to be provided by the application. For example, most components designed for multitier systems such as Tomcat or MySQL servers provide clustered modes to ensure volatile state availability through data replication as long as at least one replica is operational. Protection for applications that do not support state replication can be provided using VM-level replication techniques such as [8].

Our approach is realized by a runtime controller that monitors the system and chooses the best system configuration when a failure or recovery occurs. The controller executes in an operations center that manages the target system, and is tied to the output of a monitoring and alarm system. Centralized monitor aggregation and alarming is facilitated by many off-the-shelf products such as IBM's Tivoli, and is commonly used in commercial data centers. The algorithms we propose are deterministic, and we assume that controller availability can be ensured using traditional state machine replication. When an alarm regarding impending or actual machine failure or recovery (repair/replacement) is received from the monitoring system, the controller reconfigures the applications in the system to maintain the desired replication levels using standard virtual machine techniques. Specifically, for each VM that contains an application component, the controller can either migrate the VM to another host, or change the CPU share allocated to the VM on its current host.

The controller chooses these actions in such a way as to minimize the overall performance degradation in the case of a failure while still maintaining the desired level of replication and reliability. It has to balance several factors in doing so. Maximizing performance dictates that application components be placed close to one another to minimize the impact of network latency, but packing components too closely (e.g., on the same machine) may actually degrade performance by forcing VMs to use less CPU resources than they require. For reliability, an application requiring high levels of reliability will have to be distributed across higher resource tiers. Therefore, it ensures that redundant components of an application are located across different resource tiers at an appropriate level and thus preventing single failures from impacting multiple replicas. To achieve the required reliability level, it may be possible that the controller distributes components across a high resource level (e.g., across data centers), and thus results in poor application performance. Therefore, after producing a candidate configuration, the controller informs each application of the performance it can expect in the new configuration. If an application finds this performance level unacceptable, it can reduce its desired reliability level, and request that the controller produce another candidate configuration.

3. Metrics and Models

Our system has dual goals: high availability and good performance. We consider the system to be available when at least one replica of each component of each application is running on an operational machine, and define availability as the fraction of time the system is available over a specified time window. A replication level of at least two for each of the application's component types is necessary to avoid single points of failure. However, this is not always sufficient. If all replicas of the same type are contained within a single resource tier, e.g., a rack, then a failure of that tier can still cause application failure. Although we anticipate failure rates to decrease as one moves up the resource hierarchy, they must still be accounted for when computing application availability. Furthermore, even if a single failure does not bring down a whole application, the replication level of some component types may be temporarily reduced while the controller chooses and implements its regeneration decisions. Regeneration ensures this window of reduced replication is small, but it cannot be completely eliminated. Therefore, we allow each application to indicate to the controller its desired level of reliability by specifying a desired value for the application's "mean time between failures", or MTBF$_a$. This application MTBF can be used together with information about the system's recovery policy codified by the MTTR to calculate its availability as $\mathrm{Availability_a} = \mathrm{MTBF_a}/(\mathrm{MTBF_a} + \mathrm{MTTR})$.

For performance, we use the performance of the initial system configuration before any failures or control actions occur as the goal performance. Specifically, let $RT^g_{a,t}$ denote this goal response time for a transaction t belonging to application a and let $RT^m_{a,t}$ be the measured mean response time for this application and transaction. The performance degradation D due to resource failures can then be defined as the weighted sum of per transaction degradations, i.e.,

$$D = \sum_{a \in A, t \in T_a} \gamma_{a,t}(RT^m_{a,t} - RT^g_{a,t}) \quad (1)$$

where weights $\gamma_{a,t}$ are used to weigh the transaction according to varying importance and frequency of occurrence of the transaction. Choosing the weight as the fraction of transactions of type t in the application's workload makes D equal to a's mean response time degradation. Given these two metrics, alternative control strategies can be conveniently compared.

To quantify the performance of alternative system configurations, our approach requires application models that predict the response times of application transactions given a workload (to calculate degradation D) and the corresponding resource utilization demands in different system configurations. In this paper, we use layered queuing models [27] and the command-line version of the LQNS tool [13] as the model solver. The LQN models used, and the validation of the models, is described in [17]. Note, however, that other types of models could also be used including [6].

Given a set of MTBF values for each resource level r in the data centers and the placement of the application components on these resources, the reliability models allow the computation of the MTBF values for each application a. These values are used as constraints by the optimizer to determine the resource levels over which an application's nodes can be distributed. For the reliability models, we assume that each resource level fails independently according to a Poisson failure process with rate $\lambda_r = 1/MTBF_r$ and each failure disables all the hosts and application components it contains. If the replication level of an application drops to zero as a result of a resource failure, then the application is considered to have failed. Consider all the N_a component types of application a. For each type $n_a \in N_a$, let $r^{max}(n_a)$ be the highest resource tier such that all replicas of n_a are contained in that resource tier. From the example shown in Figure 1, if an application's database had 2 replicas hosted in DataCenter1:Rack1:Host1 and DataCenter1:Rack2:Host3, then $r^{max}(db_a) = \mathrm{DataCenter1}$. Then, only those failures at resource levels $r^{max}(n_a)$ or higher will cause the replication level of component type n_a to fall to zero and thus result in a failure of application a.

Under these assumptions, the overall failure process is also a Poisson process with rate $\sum_{r \in R} \lambda_r$. When a failure event occurs, it affects resource r with probability $\lambda_r / \sum_{r \in R} \lambda_r$, and causes application a to fail if r is such that there is at least one component type n_a with a value of $r^{max}(n_a)$ that is lower than r (i.e., $r^{max}(n_a) \leq^*_R r$). Since this condition acts to filter failure events, the application failure process is also a Poisson process with a rate given by the sum of λ_r values over those resources whose failure also causes application a to fail. Additionally, since the filtering is a function of the $r^{max}(n_a)$ values that are dependent on the exact system configuration, the Poisson process is a time-varying one whose rate changes whenever the system is reconfigured by the regeneration controller.

Using the preceding discussion, the MTBF for application a in a system configuration c is given by:

$$\mathrm{MTBF}_a(c) = \Big(\sum_{\substack{\forall r \in R \text{ s.t. } \exists n_a \in N_a \\ \text{s.t. } r^{max}(n_a) \leq^*_R r}} MTBF_r^{-1} \Big)^{-1} \qquad (2)$$

This equation makes the simplifying assumption that no additional failures occur in the time window between the first failure and the time the regeneration controller takes to decide and implement its reconfiguration actions. While this is not strictly true, it is a reasonable assumption in light of the fact that all our considered failure events occur independently and as shown in Section 6, controller think time and reconfiguration action durations are very short compared to resource MTBF values.

4 Runtime Optimizer

Upon failure or recovery events, the runtime regeneration controller chooses system configuration that maintain the applications' replication levels and desired MTBF values, and minimize any performance degradation by minimizing the degradation function in Equation 1. The minimization is carried out over the space of all possible system configurations $c \in C$, each of which specifies: (a) the assignment of each replica n_k to a physical host $c.\mathrm{host}(n_k)$, and (b) the CPU share cap $c.\mathrm{cap}(n_k)$. Due to the large parameter space with mixed discrete and continuous parameters, the optimization task is challenging. To make matters worse, the choice of a host is not just influenced by its available CPU capacity, but also by its location in the resource hierarchy in comparison with the application's other components. Even the relatively simple problem of replica assignment in a single resource tier is NP-Complete (via a reduction to the bin-packing problem), so we have to settle only for approximate solutions..

To solve this problem, we split it into two sub-problems: (a) a *search step* that chooses parameters that have an impact on the objective function, i.e., application performance, and (b) a *fit step* that chooses parameters that do not. Then, we treat the parameters considered in the fit step as the constraints, and use them as an *accept-reject* mechanism for the parameters proposed by the search. In the case of a rejection during fitting, we iterate by invoking the search step again to search for another solution, and repeat until an acceptable configuration is found. In choosing the parameters for the search step, we note that the CPU cap $c.\mathrm{cap}(n_k)$ allocated to a application component can impact the application's end-to-end performance, and so it must be included in the search.

However, choosing the machine on which to host a component can be decomposed across the search and fit steps by observing that for a given assignment of CPU caps to its components, network latency is the only other variable that

Input: W: workload
Input: c_{orig}: original config., $\{RT^g\}$: initial resp. times
Input: R: avail. resources after failure/recovery event
Output: $Acts$: sequence of reconfiguration actions
forall $a \in A$ **do**
  $\forall n_k \in N_a^k, c.\text{cap}(n_k) \leftarrow 1$
  $c.dl_a \leftarrow \min\{rl \in RL | \forall r \in R \text{ s.t. } r.rl = rl,$
    $\text{MTBF}(c[\forall n \in N_a, r^{max}(n) \leftarrow r]) > \text{MTBF}_a\}$
$CC \leftarrow \{c\}$, $c_{old} \leftarrow c$, Compute $c.D$;
while *forever* **do**
  forall $c \in CC$ **do**
    $\{c.\text{host}(n_k) | \forall a, n_k\} \leftarrow$ `Fit` (*WholeSystem*, c, A)
    if *success* **then** $FCC \leftarrow FCC \cup \{c\}$
    forall $a \in A$ **do**
      $(\{RT_{a,t}^m\}, \{\rho(n_k) | \forall n_k \in N_a^k\}) \leftarrow$ `LQNS` (W, a, c)
    Compute $c.D$, gradient $c.\nabla\rho$ w.r.t. c_{old}
  if $FCC \neq \{\}$ **then**
    $c \leftarrow \min_{c \in FCC} c.D$; **return** `Acts` ($c_{orig} \rightsquigarrow c$)
  else
    $c \leftarrow \max_{c \in CC} c.\nabla\rho$; $c_{old} \leftarrow c$; $CC \rightarrow \{\}$
    forall $a \in A$ **do**
      $c' \leftarrow c$; Inc. $c'.dl(a)$; $CC \leftarrow CC \cup \{c'\}$
      forall $n \in N_a$ **do**
        $c' \leftarrow c$
        $\forall n_k \in \text{reps}(n), c'.\text{cap}(n_k) = c.\text{cap}(n_k) - \Delta r$
        $CC \leftarrow CC \cup \{c_n\}$

Algorithm 1: Search Algorithm

impacts performance. Furthermore, according to our definition of $L(rl)$ in Section 3, the network latency is a function of the resource level rather than individual resources. Thus, the models assume the latency between two components in different racks to be the same irrespective of which rack they are in as long as both racks share the same parent in the resource hierarchy. Therefore, the choice of which resource level to distribute an application across, i.e., the *distribution level*, is determined in the search step, while the actual component placement within the distribution level is determined in the fit step. For example, a distribution level of "rack" would require placement of replicas of the same type across different hosts in the same rack, while a distribution level of "whole system" would entail placing the replicas on hosts in different data centers. The search and fit steps are described in more detail below.

Search Algorithm. The search algorithm in Algorithm 1, is a greedy algorithm that starts from the best possible configuration irrespective of capacity constraints, and iteratively degrades the configuration until it is accepted by the fit algorithm. To choose the starting configuration, we use the observation that the degradation function (response time) of an application does not decrease if additional CPU capacity is provided to one of its replicas (this is an assumption that is true for most systems). Therefore, the initial CPU caps are set to 1.0 (i.e., the entire CPU) for each replica irrespective of actual CPU availability. As the distribution level $c.dl_a$ for application a in initial configuration c, we choose the lowest resource tier that satisfies the application's MTBF_a requirement. This is because a low resource tier with lower network latency is better from a performance point of view when CPU capacity constraints are not a factor. To choose the lowest permissible resource level, we use equation 2 to compute the application MTBF when distributed across each possible resource tier r, i.e., with the value of $r^{max}(n_a)$ set to r for all the application's component types in set N_a. Then, we pick the lowest resource level for which all MTBF values are higher than the application's desired MTBF.

Input: r_p: resource tier to pack, $c.\text{cap}$: Bundle CPU capacities, B: the application and replica bundles
Output: $c.\text{host}$ - replica placements
forall $b \in B$ **do**
  if $c.dl_b = r_p.rl$ **then**
    $B \leftarrow (B - \{b\}) \cup \{\text{Replicas of } b\}$
$R_p \leftarrow \{r \in R | r \leq_R r_p\}$
forall $r \in R_p$ **do** $r.\text{cpu} \leftarrow$ Total CPU cap. of resource r
$B' \leftarrow$ `sort` ($c.\text{cap}(b) | \forall b \in B$)
forall $b \in B'$ *in decreasing order* **do**
  forall $r \in R_p$ *in order* **do**
    if $c.\text{cap}(b) \leq r.\text{cpu} \wedge (\text{reps}(\text{type}(b)) > |R_p| \vee \neg\exists b' \in B, \text{ s.t. } c.\text{host}(b') = r \wedge b.\text{type} = b'.\text{type})$ **then**
      $c.\text{host}(b) \leftarrow r$; $r.\text{cpu} \leftarrow r.\text{cpu} - c.\text{cap}(b)$
forall $r \in R$ **do** Fit($r, c, \{b \in B | c.host(b) = r\}$)

Algorithm 2: Fit Algorithm

The search algorithm explores the configuration-space using a set of "candidate configurations", or CC, which initially includes only the "best possible" configuration described above. For each configuration currently in the candidate set, the fit algorithm is invoked to try to bin-pack the nodes on physical machines using the CPU caps as the "volume" of each node. The LQNS solver is also invoked for each application to estimate response time and the actual CPU utilization $\rho(n_k)$ of each node using the chosen CPU caps and network latencies corresponding to the application's distribution level. If one or more candidate configurations provide a feasible fit as collected in set FCC in the algorithm, then the feasible configuration with the lowest performance degradation is chosen.

Otherwise, the algorithm picks the candidate configuration that maximizes a "gradient function" defined as follows.

$$\nabla\rho = \frac{-\Delta(\max_{a \in A, n_k \in N_a^k} \rho(n_k) - \max_{h \in H} \rho(h))}{\Delta D} \quad (3)$$

The gradient function is defined as the ratio of the change

in "shortfall CPU capacity" between the initial and the candidate configurations to the change in overall application performance. The shortfall CPU is defined as the difference between the CPU utilization $\rho(n_k)$ of the largest unallocated replica and the maximum CPU capacity $\rho(h)$ still available on some host in the system. The configuration that results in the greatest decrease in shortfall capacity per unit decrease in performance as compared to the current configuration is the one chosen as the next "current configuration".

The search algorithm then creates a new candidate configuration set CC by exploring single-change degradations of the chosen configuration by reducing the allowed CPU cap for the replicas of a single component in a single application by a step of Δr (set to 5% by default) or increasing the distribution level of a single application to the next higher level to provide it with access to more capacity. The algorithm is repeated with this new set of candidate configurations until the CPU capacity allocations and application distributions are sufficient for a feasible configuration to finally be found.

Upon finding a feasible configuration, the optimizer calculates the difference between the original configuration and new configuration for each replica, and returns the set of actions (migrate, capacity adjust, reinstantiate) needed to affect the change. The durations of these actions are relatively short compared to typical MTBF values, and range from a few milliseconds to a few minutes at most. Furthermore, they can be performed without causing VM downtime [7]. Therefore, the controller does not factor in any reconfiguration costs while making its decisions.

Fit Algorithm. The fit algorithm in Algorithm 2, uses a hierarchical bin packing approach to perform two tasks. First, it determines whether the node CPU caps and application distribution levels assigned by the search algorithm can be feasibly packed into the given resources. Second, it also determines actual component placement by assigning physical hosts to each replica. To do so, it initially considers each application as a single "application bundle" with volume equal to the sum of all the CPU caps of all its replicas as predicted by the queuing models. Packing is done on a resource tier by resource tier basis, starts with the "whole system" at the top of the resource hierarchy, and allocates bundles between different data centers. Subsequently, lower levels are packed, i.e., an allocation of bundles assigned to a data center across its clusters, followed by allocation across different racks in a cluster, and ending with an allocation across machines in a rack. For applications whose distribution level is equal to that of the current resource tier being packed, the algorithm breaks the application bundle into individual "replica bundles", each with a volume equal to the replica's own CPU cap $c.\mathrm{cap}(n_a^k)$. The replica bundles subsequently travel as independent units during the packing of lower level resource tiers.

Packing within a single resource tier is done using a constrained variant of a standard bin-packing algorithm. We use a constrained variant of the $n \log n$ time first-fit decreasing algorithm in which bundles are considered in decreasing order of their CPU caps, and are assigned to the first child resource tier which has enough remaining CPU capacity to fit them, and on which no other replica of same bundle exists. In the rare case that no such resource tier can be found because the number of available child resource tiers is smaller than the number of replicas of the bundle, the constraint is relaxed for that replica, and it is placed in the first resource tier on which it fits regardless of whether there is another replica of the same bundle on that resource tier.

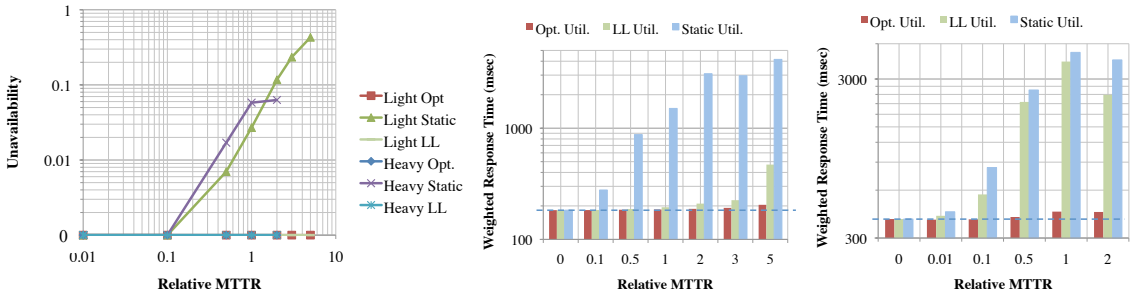
5 Simulation Results

In this section, we present simulation results using a simulator written in the Java based SSJ framework [19]. The target application for the experiments is the RUBiS online auction benchmark. We created the LQNS model using offline measurements from [17] and execute the model using transaction workload rates representing user behavior according to the "browsing mix" defined by the RUBiS test client generator.

We compare our approach (Opt) with two reference strategies: a) the Static strategy that relies on the design redundancy to tolerate failures, and b) the "least loaded" (LL) strategy that reinstantiates each failed replica (VM), in the order of decreasing CPU utilization, on the least loaded hosts. The utilization of the target host is then updated to take into account the reinstantiated VM before choosing a host for the next failed VM. Once the VMs have been reassigned, the controller reallocates the CPU capacities to the VMs on each host proportional to their measured CPU utilization with a lower bound of 10% CPU. When a host is recovered/replaced, LL migrates the original VMs running on the host before it failed from their current locations back to the host.

We evaluated the three approaches in two simulation setups. The *cluster setup* considers a local cluster of machines with identical communication delays between them. The *cloud setup* considers a resource pool distributed across two data centers, with three clusters in each data center, 3 racks in each cluster, and 4 machines in each rack. The communication delays between VMs vary depending on if they share a host, rack, cluster, or a data center. When the communication latency between two machines in a rack is D, the communication delays between machines in different racks, clusters, and data centers are $1.5D$, $2D$, and $2.5D$, respectively.

For each scenario and strategy, we ran fault injection ex-



(a) Unavailability (b) Light Workload Performance (c) Heavy Workload Performance

Figure 2. Cluster setup results

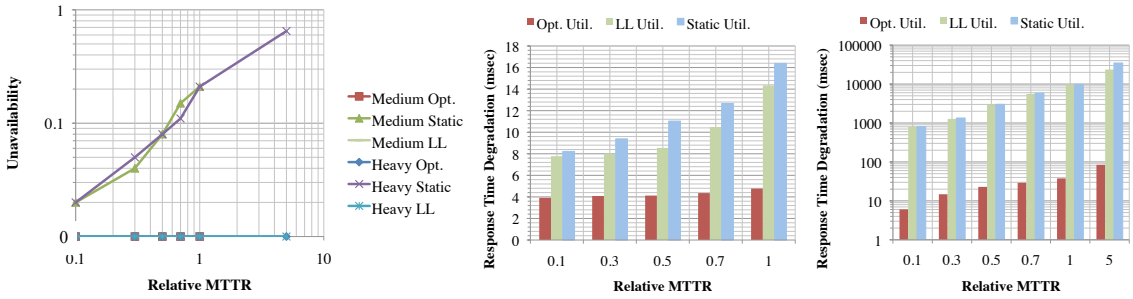
periments where host, rack, cluster, and data center failures were simulated. To make the results applicable for systems with different MTBFs and MTTRs, we report all times normalized to the MTBF, which was set to 1.0, and the MTTR values were varied over a range. Each simulation ran for a normalized time period of 10 (i.e., 10 failures per run on the average), and we repeated each experiment 10 times. For each experiment, we calculated both the availability of the system and the performance degradation.

The cluster setup consisted of 2 instances of the RUBiS application: gold and silver, where the gold instance has weight of 5 and the silver instance has weight of 1. For both instances, each of the three tiers was replicated twice. We considered two scenarios. The *light scenario* simulates an underutilized setup with an initial configuration that has each of the 12 VMs running on a separate physical host and with a workload of 60 and 120 req/sec for the gold and silver instances. The *heavy scenario* simulates a heavily utilized consolidated server environment with 8 machines, where each Apache replica (with 25% CPU capacity) shares a physical host with a Tomcat replica (with 75%) and the MySQL replicas run on dedicated hosts. The workloads are 120 and 180 req/sec for the gold and silver instances. Faults were simulated with an Poisson arrival process followed by random selection of the target host to fail. For repair, we varied the per host MTTR over a wide range from 0.01 to 5.0, indicating that repair took from 10% to 500% of the actual MTBF.

Figure 2(a) shows the unavailability of the system as a function of the relative MTTR. Both the Opt and LL strategies achieved 100% availability, while the unavailability of the Static strategy increases significantly with the relative MTTR. Since both LL and Opt regenerate VMs as soon as a failure occurs, this result is expected. In practice, both strategies may not achieve 100% availability for two reasons. First, the controllers require time to make a reconfiguration decision after a failure event and second, instantiation of new VMs is not instantaneous. During both intervals, the system may be vulnerable to additional failures. Fortunately, both windows are short compared to typical MTBF values—we have measured the VM instantiation times to be on the order of 80-90 seconds for the RUBiS MySQL instances, while the controller execution times are presented in Table 1.

Figures 2(b) and (c) show the performance degradation D of the two applications computed over the period that they are available vs. the MTTR. The initial response time of the system is indicated by the dashed line. Static performs significantly worse than the other two strategies as expected. The Opt approach has very little performance degradation even at high relative MTTR values (less than 12% in the worst case). In fact, in the heavy workload scenario, there are small improvement in the mean response times due to the fact that the optimizer was able to find a better configuration than the manually selected initial configuration. The LL results provide the most insight into the strength of the Opt controller. LL is fairly competitive with Opt in the light workload scenario, but in the heavy scenario LL performs significantly worse than Opt, and almost as poorly as Static. The reason is that in the light workload, there is enough spare capacity that reconfiguration can be performed without significantly reducing replica capacities, while that is not the case under heavy workloads. The LL controller lacks the necessary tools to make intelligent decisions about which components are bottlenecks. Instead, it makes decisions on small differences in host CPU utilizations (since all of them are high), and can end up co-locating a regenerated VM with a bottleneck resource, with great negative impact to the response time.

The cloud setup experiments consider 6 instances of the RUBiS applications, 3 gold and 3 silver. We consider two scenarios. In the *heavy scenario*, workloads of silver and gold services were 60 req/sec and 120 req/sec, respectively, while in the *medium scenario* they were 30 req/sec and 60 req/sec. Each VMs was initially allocated 80% of one CPU. The gold applications require higher availability and are thus replicated over separate clusters, while the silver



(a) Unavailability (b) Medium Workload Performance (c) Heavy Workload Performance

Figure 3. Cloud setup results

applications are replicated over racks. Failures were simulated at different levels of the hierarchy (i.e., data center, cluster, rack, host) with different failure and repair rates. Specifically, if the MTBF and MTTR on the host-level are M_f and M_r, then at the rack, cluster, and data center levels they are $4M_f$ and $4M_r$, $16M_f$ and $16M_r$, and $160M_f$ and $160M_r$, respectively. For repair, we varied the per host relative MTTR from 0.1 to 1, indicating that repair took from 10% to 100% of the actual MTBF. The LL strategy was modified to account for the different levels of resource hierarchies.

Figure 3(a) demonstrates that with our given MTTR and MTBF rates, the regeneration strategies can ensure high availability, while the Static strategy fails to do so. Figures 3(b) and (c) show the advantage of Opt over LL, particularly under the heavy workloads. Note that the weighted response times for Opt are in the range of 6 to 37 msec and as a result, some of them do not show in the plot. The results show that in some cases LL does not perform much better than Static. This is because if a set of hosts (a whole rack, cluster, or data center) fails and the failed hosts contain the VMs of the silver applications, then it may be better to do nothing (i.e., Static) than reallocating those VMs in LL.

Finally, Table 1 shows the measured execution times of the Opt controller both in the cluster and cloud scenarios. As shown, the execution time depends on the target system configuration and workload, but is still reasonable even for cloud computing scenarios. Improving the scalability of the optimizer remains an area for improvement in our future work.

Cluster			Cloud		
MTTR	Light	Heavy	MTTR	Medium	Heavy
0.1	2.5	51	0.1	31	81
0.5	3.5	55	0.5	79	150
1.0	4.3	59	0.7	99	174
2.0	5.5	59	1.0	119	200

Table 1. Controller Exec. Time (sec)

Overall, the results indicate that the Opt controller correctly navigates around bottleneck situations using its queuing model predictions. This feature makes our proposed approach especially suitable for use in the growing number consolidated server environments which typically have much higher utilizations than dedicated hosting setups.

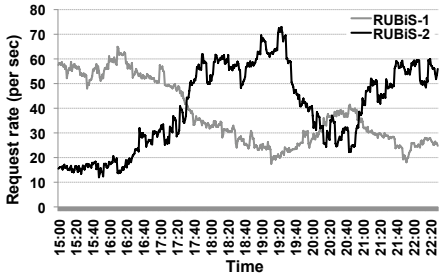


Figure 4. Workload

6 Fault Injection Results

Next we present experimental fault injection results whose goal is to demonstrate how our technique compares with the static and least loaded strategies in terms of application performance degradation when subjected to actual failures and realistic workloads. The target application used in our experiments is the 3-tier version of RUBiS. The resource pool in our experiments consists of 10 machines divided over two racks (5 in each). Each physical host is equipped with an Intel Pentium 4 1.80GHz processor, 1 GB RAM, and a 100 Mb Ethernet interface. We used the open-source version of the Xen 3.2.0 to build the virtualization environment. The Linux kernel 2.6.18.8 was installed as a guest OS in each Xen user domain. Apache 2.0.54, Tomcat 5.0.28, and MySQL 3.23.58 were used as the web server, servlet container, and database server respectively. Each replica was installed in its own virtual machine. We ran the regeneration controller on a machine with 2 Intel Xeon 3.00GHz processors and 4 GB RAM.

Two instances of RUBiS are used for these experiments, referred to as RUBiS-1 and RUBiS-2. We drive the target applications using workloads generated based on the

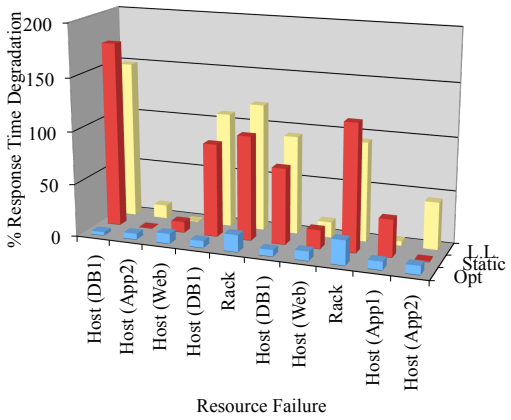


Figure 5. Response time change (%) after failure and reconfiguration

Web traces from the 1998 World Cup site [2] and the traffic traces of an HP customer's Internet Web server system [12] and scaled to the range of request rates that our experimental setup can handle. Figure 4 shows these scaled workloads for the two RUBiS applications from 15:00 to 22:30. RUBiS-1 uses the scaled World Cup workload profile and RUBiS-2 uses the scaled HP workload profile. The two applications are replicated with two replicas for each tier, and the replicas are placed on machines in the different racks. Thus, a total of 12 VMs were deployed on the 10 machines. In the initial configuration, the VMs hosting the Tomcat and MySQL replicas were allocated 80% of the CPU capacity and the VMs hosting Apache replicas were allocated 40% of the CPU capacity. Thus, only the VMs hosting the Apache replicas could be co-located on the same physical machine in the initial configuration. For the Static strategy, the placement and the capacity allocation of the VMs remains the same throughout the experiment, while the Opt and LL strategies adjust the location and capacity allocation of the VMs based on the workload at the time of the failure.

We emulate the failures on individual machines or racks (i.e., a correlated failure of all machines in a rack due to a common cause such as power source or switch/router). For each experiment, we pick a random location in the timeline for the emulated failure. We measure the mean response time for a period of 5 minutes before the emulated failure and 5 minutes after the emulated failure and reconfiguration. The same failure location in the timeline is emulated for each strategy, that is, each strategy is subject to the same workload at the time of the failure.

Across all the emulated failure scenarios and across these two applications, the average performance degradation for our approach was 9.5%, while for the Static and LL approaches the degradations were 46% and 47%, respectively. The workload intensity is relatively small, and thus, the gap between Static and LL approaches is very small because a new configuration after a failure can still deal with the workload without reallocations of VMs. Due to this fact, the Static approach even outperforms the LL approach in some cases. Figure 5 illustrates the difference (%) between the response times before and after failure (and reconfiguration) for one set of emulation points for RUBIS-1. The workloads at the different emulation points vary between 23 and 56 requests/sec and thus, the different points are not directly comparable. However, the results show that our approach consistently provides graceful degradation even when whole racks fail.

7 Related Work

Dynamic creation of of new replicas to account for failures has been used before. For example, [23] uses regeneration of new data objects to account for reduction in redundancy and the Google File System [14] similarly creates new file "chunks" when the number of available copies is reduced below a threshold. Even commercial tools such as VMWare High Availability (HA) [26] allow a virtual machine on a failed host to be reinstantiated on a new machine. However, the placement of replicas becomes especially challenging when they are components in a multitier application. Recent work on performance optimization of multitier applications (e.g., [25, 4, 9, 17]) address the performance impact of resource allocation on such multitier applications, but to our knowledge our work is the first to combine performance modeling of multitier applications with availability requirements and dynamic regeneration of failed components.

The tradeoff between availability and performance is always present in dependability research since increasing availability (by using more redundancy) typically increases response time. Examples of work that explicitly address this issue include [10] and [24], both of which consider the problem of when to invoke a (human) repair process to optimize various metrics of cost and availability defined on the system. In both cases, the "optimal policies" that specify when the repair was to be invoked (as a function of system state) were computed off-line through solution of Markov Decision process models of the system.

Finally, the combination of fault tolerance and timeliness (hard or soft real-time) properties has been traditionally considered in systems research [20, 18]. Fault-tolerant real-time systems such as Mars [18] use static replication of processing tasks to ensure reliability in the real-time system. The real-time scheduling community has a long history of adding fault tolerance to real-time task execution by either scheduling a task on multiple processors or by reserving slots in the schedule for repeated execution in case of a failure [20, 1, 21]. Researchers in the middle-

ware community have also addressed the combination of fault tolerance and timeliness in platforms such as CORBA and Java RMI, which requires balancing the often conflicting requirements associated with implementing the two attributes [16, 22]. However, the specific techniques (e.g., admission control, thread and request priorities, deadline based scheduling) used are very different from the course-grained VM provisioning used in our work and the performance goal of meeting the mean response time thresholds for a collection of applications sharing a resource pool is different from meeting the absolute (hard or soft) request deadlines for one application.

8 Conclusions

In this paper, we have examined how virtual machine technology can be used to provide enhanced solutions to the classic dependability problem of ensuring high availability while maximizing performance on a fixed amount of resources. We use component redundancy to tolerate single machine failures, virtual machine cloning to restore component redundancy whenever machine failures occur, and smart component placement based on queuing models to minimize the resulting performance degradation. Our simulation results showed that our proposed approach provides better availability and maximum throughput than classical approaches.

References

[1] K. Ahn, J. Kim, and S. Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proc. PRFTS*, page 98, 1997.

[2] M. Arlitt and T. Jin. Workload charaterization of the 1998 world cup web site. In *HP Technical Report, HPL-99-35*, 1999.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Wareld. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[4] M. Bennani and D. Manesce. Resource allocation for autonomic data centers using analytic performance models. In *Proc. ICAC*, pages 217–228, 2005.

[5] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proc. Middleware*, 2003.

[6] S. Chen, K. Joshi, M. Hiltunen, R. Schlichting, and W. Sanders. Gradient-based Predictive Models of Multitier Systems. In *Proc. PMCCS*, 2009

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, 2005.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. NSDI*, pages 161–174, 2008.

[9] I. Cunha, J. Almeida, V. Almeida, and M. Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Proc. IM*, pages 129–138, 2007.

[10] H. de Meer and K. S. Trivedi. Guarded repair of dependable sys. *Theoretical Comp. Sci.*, 128:179–210, 1994.

[11] J. Dean. Software engineering advice from building large-scale distributed systems. Stanford CS295 class lecture. http://http://research.google.com/people/jeff/stanford-295-talk.pdf, 2007.

[12] J. Dilley. Web server workload charaterization. In *HP Technical Report, HPL-96-160*, 1996.

[13] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. Performance analysis of distributed server systems. In *Proc. Intl. Conf. Software Quality*, pages 15–26, 1996.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. SOSP*, 2003.

[15] J. R. Hamilton. An architecture for modular data centers. In *Proc. Innovative Data Sys. Research*, pages 306–313, 2007.

[16] J. He, M. Hiltunen, M. Rajagopalan, and R. Schlichting. QoS customization in distributed object systems. *Software–Practice and Experience*, (33):295–320, 2003.

[17] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments,. In *Proc. ICAC*, pages 23–32, 2008.

[18] H. Kopetz and W. Merker. The architecture of Mars. In *Proc. FTCS*, pages 274–279, 1985.

[19] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: a framework for stochastic simulation in Java. In *Proc. Winter Simul. Conf.*, pages 234–242, 2002.

[20] A. Liestman and R. Campbell. A fault-tolerant scheduling problem. *IEEE Trans. SE*, SE-12(11):1089–1095, 1986.

[21] D. Mosse, R. Melhem, and S. Ghosh. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Trans. SE*, SE-29(8):752–767, 2003.

[22] P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. Mead: Support for real-time fault-tolerant corba. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[23] C. Pu, J. Noe, and A. Proudfoot. Regeneration of replicated objects: A technique and its eden implementation. In *Proc. Int. Conf. on Data Engineering*, pages 175–187, 1986.

[24] K. G. Shin, C. M. Krishna, and Y.-H. Lee. Optimal dynamic control of resources in a distributed system. *IEEE Trans. SE*, 15(10):1188–1198, Oct 1989.

[25] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. SIGMETRICS*, pages 291–302, 2005.

[26] VMWare. Vmware high availability (ha), restart your virtual machine. Accessed May 2009. World Wide Web. http://www. vmware.com/products/vi/vc/ha.html.

[27] C. M. Woodside, E. Neron, E. D. S. Ho, and B. Mondoux. An "active server" model for the performance of parallel programs written using rendezvouz. *J. Systems and Software*, pages 125–131, 1986.