Performance Aware Regeneration in Virtualized Multitier Applications

Kaustubh Joshi and Matti Hiltunen
AT&T Labs Research
180 Park Ave
Florham Park, NJ 07932
Email: {kaustubh,hiltunen}@research.att.com

Gueyoung Jung
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
Email: gueyoung.jung@cc.gatech.edu

***Abstract*—Virtual machine technology enables highly agile system deployments in which components can be cheaply moved, cloned, and allocated controlled hardware resources. In this paper, we examine in the context of multitier Enterprise applications, how these facilities can be used to provide enhanced solutions to the classic problem of ensuring high availability without a loss in performance on a fixed amount of resources. By using virtual machine clones to restore the redundancy of a system whenever component failures occur, we achieve improved availability compared to a system with a fixed redundancy level. By smartly controlling component placement and co-location using information about the multitier system's flows and predictions made by queuing models, we ensure that the resulting performance degradation is minimized. Simulation results show that our proposed approach provides better availability and significantly lower degradation of system response times compared to traditional approaches.**

I. Introduction

Current trends in system and data center design cast an interesting perspective on the role of redundancy and repair in dependably operating the multitier applications that power most of the world wide web. On one hand, multitier systems are increasingly running on large numbers of cheap, less reliable commodity components, thus leading to a decrease in the system's mean time between failures (MTBF) numbers. For example, in [1], Google reported an average of 1000 node failures/yr in their typical 1800 node cluster for a cluster MTBF of 8.76 hours. On the other hand, skilled manpower is quickly becoming the most expensive resource in the mix, thus encouraging data center operators to achieve economies of scale by batching repairs and replacement, and increasing mean times to repair/replacement (MTTR) in the process. In fact, there have been attempts at portable "data-center in a box" designs (e.g., [2]) that contain tightly packed individual components that are completely non-serviceable, i.e., with an infinite MTTR.

These trends imply that multitier systems will increasingly operate in environments in which parts of the infrastructure are in a failed state for most of their operation. While current designs often replicate the software components comprising each application tier, to truly ensure high availability in the above conditions, the failed components must eventually be replaced, and the level of redundancy must be high enough to tolerate additional failures while replacement takes place. However, maintaining significant redundancy is challenging. Active replication may be too expensive in cases such as when dynamic state has to be replicated, or when expensive software licenses are involved. On the other hand, reducing effective time-to-repair by maintaining standby spare resources that can be quickly deployed is inefficient because the spares represent unutilized resources that could otherwise have been used to increase the capacity of the target system.

In this paper, we investigate an alternative solution that effectively utilizes all system resources (i.e., no standby spares) while providing high availability with limited levels of replication. Specifically, when a hardware resource fails or is likely to fail, we regenerate the affected software components and deploy them on the remaining resources and proactively avoid a system failure. The basic idea of dynamically creating new replicas to account for failures is not new. For example, [3] and [4] use regeneration of new data objects and file "chunks", respectively, to account for reduction in redundancy. Even commercial tools such as VMWare High Availability (HA) [5] allow a virtual machine on a failed host to be reinstantiated on a new machine. The approach can make do with far less redundancy than a static replacement oriented design. However, because regenerated components share the remaining computing resources, the approach can have a significant impact on a system's performance if not managed correctly.

In particular, the placement of replicas becomes challenging when they are components in a multi-tier application. In such applications, poor placement 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 are being shared (e.g., in a cloud computing environment), the problem becomes even more complex. Recent work on performance optimization of multitier applications (e.g., [6], [7], [8], [9]) addresses the performance impact of workload variations and resource allocation on such multitier applications, but to our knowledge our work is the first one to combine these two lines of work.

Specifically, we propose to reconfigure a set of multitier applications, hosted on shared resources, in reaction to failures or impending failures in order to maintain a user-defined level

of redundancy for as long as possible, and to do so in a way that minimizes degradation in the applications' responsiveness. Note that in a bid to preserve performance, we increase the mandate of our solution from simply regenerating failed components to complete proactive redeployment of the entire system including those parts that are not impacted by the failure. Traditionally, performing extensive system reconfiguration has had costs, and previous dependability literature such as [10] and [11] has addressed the issue by proposing control policies to intelligently schedule reconfiguration for least negative impact. In this work however, we circumvent the issue altogether by using virtual machine technology to enable cheap and fast application reconfiguration and migration without incurring significant downtime or overhead [12].

Our solution is deployed through an online controller that uses queuing models we have previously developed in [9] to predict the performance and resource utilization of the multi-tier applications, and uses a bin-packing-based optimization technique to produce target configurations that exhibit the least amount of performance degradation. By using utility functions to prioritize applications, it can also manage a set of applications that have varying importance and SLAs.

II. Architecture

We begin by defining the class of applications considered, and their computing environment. We consider a consolidated data-center type environment with a pool of physical resources H and a set of multi-tier applications A. Although typical data-center environments have multiple pools of resources (e.g., servers, disks, I/O channels), we focus only on CPUs in this paper. Furthermore, we assume a cluster of identical physical hosts. Nevertheless, the techniques we propose are general and can be applied to manage other resource pools as well.

Each application a consists of a set N_a of component types (e.g., web server, database), and for each component type n, a desired replication level is provided by $\mathrm{reps}(n)$. To avoid single points of failure, the replication level for each component must be at least 2. Each replica n_k executes in its own Xen virtual machine [13] on some physical host, and 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. Additionally, each application a is assumed to support multiple transaction types $T_a = \{t_a^1, \ldots, t_a^{|T_a|}\}$. For example, the RUBiS [14] auction site benchmark used in our testbed has 26 different transactions that correspond to login, profile, browsing, searching, buying, and selling. The workload w_a for the application can then 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 $W = \{w_a | a \in A\}$.

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. 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 [15].

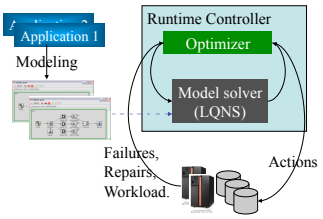


Fig. 1. Approach overview

The overall architecture of our approach is outlined in Figure 1. It consists of a Runtime Controller that executes in an operations center that manages the target cluster, 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 replication. When an alarm regarding impending or actual machine failure or recovery (repair/replacement) is received from the monitoring system, the controller reconfigures the application to maintain the desired replication levels using standard virtual machine control knobs provided by most VM engines. 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 maximizes the overall performance in case of a repair/replacement event. To avoid common mode failures, the controller also ensures that redundant components of an application are not located on the same physical machine. To achieve these objectives, the Runtime Controller relies on a model solver and application models developed offline to compare different deployment alternatives as described next.

III. Metrics and Models

a) Metrics: As indicated earlier, 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. For performance, we use the initial system configuration before any failures or control actions occur as a goal that should be achieved after a failure event has occurred. Specifically, let $RT_{a,t}^g$ denote this goal response time for a transaction t belonging to application a and let $RT_{a,t}^m$ be the measured mean response time for this application and transaction. The

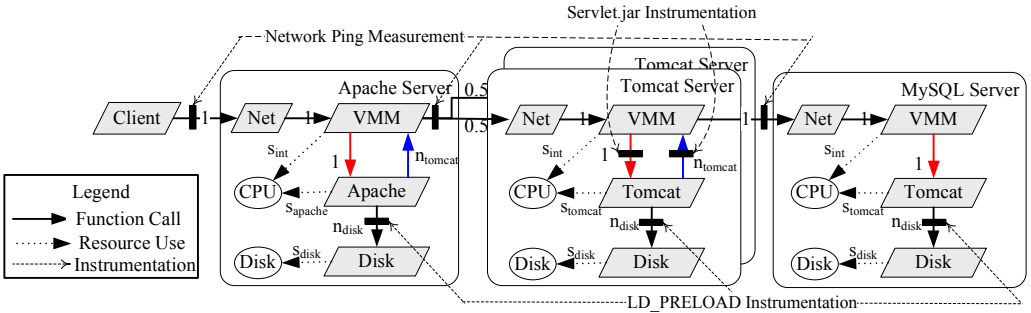


Fig. 2. Layered Queuing Network model for RUBiS

performance degradation D due to host 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.

b) Application Modeling: Application models that accurately predict the performance of a configuration given a workload (to calculate degradation D) and the corresponding resource utilization demands are an important building block for our approach. However, consolidated server environments under resource constraints can be difficult to model because CPU utilizations can become high and CPU allocations to the different tiers of an application can be very different. Thus, blocking phenomena that are not significant in well-provisioned environments, e.g., a bottleneck due to the blocking of front-end software threads by a highly overloaded back-end server, must be explicitly modeled. Because of their ability to do so, we chose layered queuing models [16] as the basis of our work and use the LQNS tool [17] as a black-box model solver.

Figure 2 shows a high-level overview of the model for RUBiS, our example application. The model includes tasks (depicted by parallelograms) that represent software components, and queues that represent hardware resources (depicted by circles). Tasks use hardware resources (depicted by dotted arrows with the label specifying the service time) and make synchronous calls to other tasks (depicted by solid arrows with the label specifying the average number of calls made). An additional complication is that the effects of the virtualization environment need to be modeled. That is done by a VMM task that represents the hypervisor, and generates CPU demand for any I/O requests that flow through it.

The models are parameterized based on measurements done in a pre-deployment training phase. For Java servlet based applications, the parameter collection is fully automated, and does not require any instrumentation in the application code. During this phase, each application is measured in isolation with only a single replica per component, and is subjected to a workload that consists of a single transaction at a time. Multiple repetitions are done to compute mean values of the parameters. The solid black boxes in Figure 2 represent the points where measurements are made. Additional details about the model and its validation against experimental measurements can be found in [9].

IV. Runtime Optimizer

Upon impending or actual failure or recovery events, the runtime controller chooses system configurations that maintain the applications' replication levels 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 $c.\text{host}(n_k)$ in a way that ensures that no two replicas of the same component share a physical host, and (b) the CPU share cap $c.\text{cap}(n_k)$. Due to the large parameter space with mixed discrete and continuous parameters, the optimization task is challenging. Even the problem of replica assignment is NP-Complete (via a reduction to the bin-packing problem), so we have to settle only for approximate solutions. However, an approximate solution purely by traditional means, e.g., gradient search, is difficult because of the mixed parameter types.

To solve this problem, we split it into two sub-problems: a) selecting an application "configuration" by choosing the replica CPU caps, a problem with a continuous parameter space, and b) determining the optimal component placement for a given application configuration, which involves discrete optimization. To integrate the two solutions efficiently, we leverage two observations: a) 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), and b) reallocating a replica to another host without changing its CPU capacity does not change its response time or that of the application. Due to observation (b), the component placement optimization can simply act as an *accept-reject* constraint for each candidate configuration generated in the configuration optimization. If the "optimal" placement can fit the required components into the available resources, then the application configuration is accepted. Otherwise, it is rejected. The optimization algorithm is shown as Algorithm 1, and is explained in detail below.

Application Configuration. Due to observation (a), if one starts off with CPU cap of 1.0 (i.e., the entire CPU) for each

Input: W: the workload
Input: c_{orig}: the original config., $\{RT^g\}$: the initial resp. times
Input: H: available hosts after the failure/recovery event
Output: sequence of reconfiguration actions
forall $a \in A, n \in N_a$ **do** $\forall n_k, c.\text{cap}(n_k) \leftarrow 1$
forall $a \in A$ **do**
  $(\{RT^m_{a,t}\}, \{\rho(n_k) | \forall n_k \in N^k_a\}) \leftarrow$ `LQNS` (W, a, c)
Compute D
while *forever* **do**
  $\{c.\text{host}(n_k) | \forall a, n_k\} \leftarrow$ `ConstraintBinPack` (H, c)
  if *success* **then return** `Actions` $(c_{orig} \rightarrow c)$ **forall** $a \in A, n \in N_a$ **do**
    $c_{new} \leftarrow c$
    forall $n_k \in \text{reps}(n)$ **do** Reduce $c_{new}.\text{cap}(n_k)$ by Δr
    $(\{RT^m_{a,t}\}, \{\rho(n_k)\})_{new} \leftarrow$ `LQNS` (W, a, c_{new})
    Compute D_{new}, $\nabla\rho$
    if $\nabla\rho > 0 \vee \nabla\rho$ *is max so far* **then**
      $(c, \{\rho(n)\})_{opt} \leftarrow (c, \{\rho(n)\})_{new}$
    if $\nabla\rho > 0$ **then** skip to **EndRefit**
  EndRefit: $(c, \{\rho(n)\}) \leftarrow (c, \{\rho(n)\})_{opt}$

Algorithm 1: Optimal configuration generation

Input: H: set of available hosts, c.cap: CPU capacities
Output: c.host - replica placements
forall $h \in H$ **do** $h.\text{cpu} \leftarrow 0$
$R \leftarrow$ `sort` $(c.\text{cap}(n_k) | \forall a \in A, n \in N_a, k \in [1 \ldots \text{reps}(n)])$
forall $r \in R$ *in decreasing order* **do**
  forall $h \in H$ *in order* **do**
    if $c.\text{cap}(r) \leq h.\text{cpu} \wedge (\text{reps}(\text{type}(r)) > |H| \vee \neg\exists r' \in R,$ *s.t.* $c.\text{host}(r) = h \wedge r.\text{type} = r'.\text{type})$ **then**
      $c.\text{host}(r) \leftarrow h$; $h.\text{cpu} \leftarrow h.\text{cpu} - c.\text{cap}(r)$

Algorithm 2: Constrained Bin Packing

replica irrespective of actual CPU availability, the degradation function would be the lowest, and monotonically increase with decreasing CPU caps. Therefore, no local maximums exist for this function, and the application configuration optimization algorithm uses a text-book gradient search algorithm to select CPU caps, terminating as soon as an acceptable configuration that fits in the actually available hosts is found. The LQNS solver is invoked for each application to estimate response time and the actual CPU utilization $\rho(n_k)$ of each node. The bin packer is then invoked to try to place the nodes on the available machines using the predicted CPU utilizations as the "volume" of each node. If the bin packing is unsuccessful, the algorithm re-evaluates all possible single-change degradations of the current 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). The algorithm then picks that degraded configuration that provides the maximum reduction in overall CPU utilization for a unit increase in degradation, or gradient, which is defined as:

$$\nabla\rho = \frac{\sum_{a \in A, n_k \in N^k_a} \rho_{\text{new}}(n_k) - \rho(n_k)}{D_{\text{new}} - D} \quad (2)$$

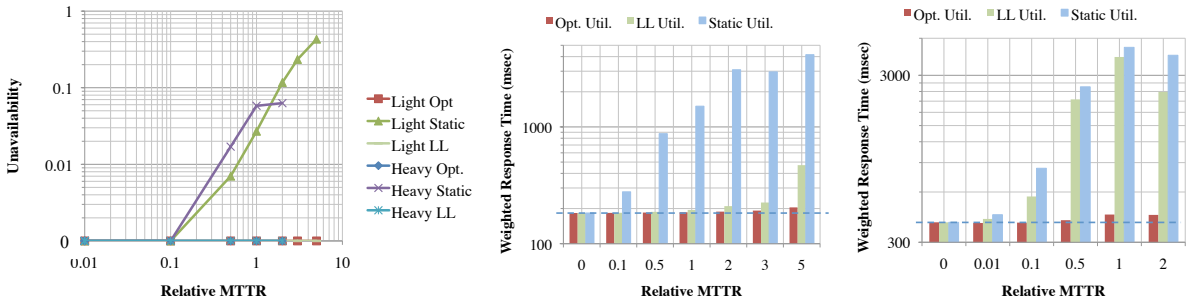
The whole process is repeated again until the bin packing succeeds. Upon success, the optimizer calculates the difference between the original configuration and new configuration for each replica, and returns the set of actions (migrate, capacity adjust, reinstantiate) needed to affect the change. The discretionary actions, i.e., capacity adjustment and migration, are relatively cheap 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 [12]. Therefore, the controller does not factor in any reconfiguration costs while making its decisions.

Constrained Bin Packing. Component placement is performed using bin packing, which serves two purposes. First, it determines whether the allocated CPU caps of all the replicas from all applications fit into the available CPU capacity, and second, it also determines to which physical host to assign each replica such that replica independence, i.e., no replicas of the same component on the same host, is maintained as far as possible. Unconstrained bin packing has been studied extensively in the literature, with several known efficient approximation algorithms.We use a constrained variant of the $n \log n$ time first-fit decreasing algorithm as shown by Algorithm 2 in which the replicas are considered in decreasing order of their CPU cap, and are assigned to the first host which has enough remaining CPU capacity to fit them, and on which no other replica of same component exists. In the rare case that no such host can be found because the number of available hosts is smaller than the replication level of the component, the constraint is relaxed for that replica, and it is placed on the first host on which it fits regardless of whether there is another replica of the same component on that host. The results of the unconstrained algorithm are guaranteed to be within 22.22% of the optimal solution [18]. While we cannot make such guarantees for our constrained version, the results from Section V show excellent performance in practice.

V. Results

We have implemented the configuration optimizer and integrated it with a previously developed runtime framework for performing adaptive VM reconfiguration in response to dynamically changing workloads [9]. However, to compare the impact of the technique on long term availability, we present simulation results across several hundred failures using a simulator written in the Java based SSJ framework [19].

c) Strawman Approaches: We compare the approach with two reference strategies: a) a do-nothing "static" strategy that does not adaptively reconfigure the system, but relies on the design redundancy to tolerate failures, and b) a strawman "least loaded" strategy (LL) that reinstantiates the "failed" VMs running on the failed host. To choose the target hosts, LL considers each failed VM in decreasing order of measured CPU utilization, and reinstantiates it on the least loaded host. 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. 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. 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 deploying an LL controller, runtime measurements of the per-host and



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

Fig. 3. Experimental Results

VM utilizations are needed. During simulation, we use the LQNS models to produce these measurements for the deployed configurations.

d) Target System Setup: The target application for the experiments is the widely used RUBiS online auction benchmark [14] from Rice University. The J2EE application runs on standard off-the-shelf components - an Apache Web server, Tomcat App server, and MySql database, and provides 26 distinct user transactions. We created the LQNS model using offline measurements from [9] and execute it using transaction workload rates representing real user behavior according to the "browsing mix" defined by the RUBiS test client generator. Our simulation setup consisted of 2 instances of the RUBiS application, each with a different workload and priority, with a "gold" instance with a weight of 5 times as much as the "silver" instance. For both instances, each of the three tiers was replicated twice. We considered two different scenarios. The "light" scenario simulates a classical, 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 requests/sec for the gold and silver instances. The "heavy" scenario simulates a more heavily utilized consolidated server environment with each Apache replica (with 25% CPU capacity) sharing a physical host with a Tomcat replica (with 75%). The MySQL replicas run on dedicated hosts constituting a total of 8 physical machines. The workloads in this setup are higher at 120 and 180 requests/sec for the gold and silver instances.

e) Simulation Setup: For each scenario and strategy, we ran fault injection experiments where host failures were simulated with an Poisson arrival process followed by random selection of the target host to fail. To make the results applicable for systems with different MTBFs, we report all times normalized to the MTBF, which was set to 1.0. For repair, we varied the per host mean time to repair (MTTR) over a wide range from 0.01 to 5.0, indicating that repair took from 10% to 500% of the actual MTBF. It is our belief that all ratios in this range are realistic depending on system size. 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 mean response time across both applications, i.e., the per-transaction response time weighted by the fractional rate of the transaction. The response time degradation of the gold application was multiplied by its priority of 5. The time spent by the controller in making a decision also impacts the availability because, during this time, the impact of an additional failure is the same as that with the static strategy. However, because it is not possible to normalize this time without choosing a value for the MTBF, we do not factor these times into the availability and report them separately.

f) Availability: Figure 3(a) shows the unavailability of the system under all strategies and workloads as a function of the MTTR. As is seen, both the Opt strategy (our approach) and the LL strategy achieved 100% availability during the simulation runs, while the unavailability of the static strategy increases significantly with the relative MTTR. Adding additional replicas can improve the availability provided by the static strategy, but the LL and Opt results show that doing so is not needed for improving system availability even with large relative MTTR values. Since the LL and Opt. strategies both 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 I. Practically, the likelihood of additional non common-mode failures during those intervals is low.

g) Response Time Degradation: Figures 3(b) and (c) show the degradation of the sum of the mean response times of both applications computed over the period that they are available vs. the MTTR. The results with an MTTR of 0 indicate the initial response times of the system. These are identical for all three strategies, and are also projected by the dashed line. As is seen in the figures, the static strategy performs significantly worse than the other two strategies as expected. The Opt. approach is the best of the three, with very little performance degradation even at high MTTR values (less than 12% in the worst case). In fact, in the heavy workload scenario, there is a small improvement of 3.76 and 1.69 msec in the mean response time for the 0.01 and 0.1 MTTR experiments respectively. This is due to the fact that the

optimizer was able to find a better configuration than the initial configuration, which was selected manually, for that scenario.

However, it is the LL results that provide the most insight into the strength of the Opt. controller. Although it always performs worse than Opt., LL is fairly competitive with the Opt. controller in the light workload scenario for all except the largest MTTR value. However, the situation is dramatically different when the heavy scenario is considered. Here, the LL controller performs significantly worse than Opt, and almost as poorly as the static strategy. The reason is that in the light workload, there is enough spare capacity that reconfiguration can be performed without significantly reducing replica capacities below what is needed by the application. That is not the case under heavy workload, and 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 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.

	Light		Heavy	
MTTR	Opt.	Static	Opt.	Static
0.1	2.53	0.38	51.33	0.81
0.5	3.50	0.45	55.32	0.82
1.0	4.31	0.52	59.42	0.87
2.0	5.51	0.66	58.56	0.88

TABLE I
CONTROLLER EXECUTION TIME IN SECONDS

h) Controller Execution Times: Finally, Table I shows the measured execution time of the Opt. controller in seconds. The execution time also includes an extra call to the LQNS solver to compute the response time degradation of the resulting target configuration which is not required in a real deployment. Therefore, as a baseline, the results for the static configuration, which has a do-nothing controller and the additional LQNS execution, are also shown. From the results, two points are worthy of note. First, the execution times, while not high enough to cause an availability issue in practice, are still not insignificant. Second, the execution time, which mostly comprises the LQNS solver execution time, depends significantly on the target system configuration. In particular, under conditions of high utilization where multiple CPU capacity reductions may be required, the execution time is higher because of a larger number LQNS solver executions during the gradient search. As can also be seen by comparing the light and heavy workload results for the static controller, another contributing factor is that each LQNS solver execution also takes longer to converge in high utilization environments. Improving the scalability of the optimizer remains the biggest area for improvement in future work.

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

[2] J. R. Hamilton, "An architecture for modular data centers," in *Proc. of the Conf. on Innovative Data Sys. Research*, 2007, pp. 306–313.

[3] C. Pu, J. Noe, and A. Proudfoot, "Regeneration of replicated objects: A technique and its eden implementation," in *Proc. 2^{nd} Int. Conf. on Data Engineering*, 1986, pp. 175–187.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. 19th ACM SOSP*, 2003.

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

[6] 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*, 2005, pp. 291–302.

[7] M. Bennani and D. Manesce, "Resource allocation for autonomic data centers using analytic performance models," in *Proc. 2^{nd} Intl. Autonomic Computing Conference*, 2005, pp. 217–228.

[8] I. Cunha, J. Almeida, V. Almeida, and M. Santos, "Self-adaptive capacity management for multi-tier virtualized environments," in *IM'07: Proc. 10^{th} IFIP/IEEE Intl. Symp. on Integrated Network Management*, 2007, pp. 129–138.

[9] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Generating adaptation policies for multi-tier applications in consolidated server environments,," in *Proc. 5th IEEE Intl. Conf. on Autonomic Computing*, June 2008, pp. 23–32.

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

[11] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "Optimal dynamic control of resources in a distributed system," *IEEE Trans. on Software Eng.*, vol. 15, no. 10, pp. 1188–1198, Oct 1989.

[12] 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. USENIX NSDI*, 2005.

[13] 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. 19th SOSP*, 2003, pp. 164–177.

[14] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proc. 4th Intl. Middleware Conf.*, 2003.

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

[16] 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*, pp. 125–131, 1986.

[17] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside, "Performance analysis of distributed server systems," in *Proc. 6th Intl. Conf. on Software Quality*, 1996, pp. 15–26.

[18] E. G. C. Jr., G. Galambos, S. Martello, and D. Vigo, *Du, D.Z., Paradalos, P.M., eds.: Handbook of Combinatorial Optimization*. Kulwer, 1998, ch. Bin Packing Approx. Algorithms: Combinatorial Analysis.

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