# Providing Dependability and Performance in the Cloud: Case Studies

Nikolaus Huber[1], Fabian Brosig[1], Nicholas Dingle[3], Kaustubh Joshi[2], and Samuel Kounev[1]

[1] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{fabian.brosig, nikolaus.huber, samuel.kounev}@kit.edu
[2] AT&T Labs Research, NJ, USA
kaustubh@research.att.com
[3] School of Mathematics, University of Manchester, Manchester M13 9PL, UK
nicholas.dingle@manchester.ac.uk

**Abstract** Cloud Computing promises a variety of opportunities but also brings up several challenges. The three case studies presented in the following are examples on how challenges in the field of capacity management, dependability, and scalability can be addressed and how opportunities of Cloud Computing can be leveraged to, e.g., maintain performance requirements or to increase dependability.

## 1 Introduction

As discussed in Chapter X[4], Cloud Computing has several challenges and opportunities. The increased flexibility and the shared resources cause challenges like security or performance issues, to mention only some examples. However, the increasing flexibility provides also opportunities like higher availability and fault tolerance, resilience to attacks, or improved resource efficiency.

In this chapter we present three case studies as examples on how the previously mentioned challenges can be addressed and how the opportunities can be used to add value to systems running in Cloud Computing environments. The first two case studies are approaches on managing performance and dependability in Cloud Computing environments. The third case study is a scalability study of two different tools for performance analysis in Cloud Computing environments. For related work and state-of-the-art on approaches for resilience assessment and managing dependability and performance, the reader is referred to Chapter X.

In Section 2, we demonstrate how prediction techniques based on performance models can be used to maintain the service-level agreements (SLAs) while using available resources efficiently. The approach uses the Palladio Component Model [3] and its simulator to predict service response times and resource utilizations. Section 3 presents an architecture and algorithm on balancing the

[4] "Providing Dependability and Resilience in the Cloud: Challenges and Opportunities"

trade-off between performance and dependability. It uses performance and availability models to react to changes in the underlying infrastructure which are results of failures or upgrades. The hierarchical optimization algorithm extends queuing models to balance the needs of availability and performance. Several scenarios show the applicability of this approach even in a cloud scenario with different data centers. Finally, we present a case study on the computational and communication scalability of a Cloud Computing environment by transferring two HPC applications to a Cloud Computing environment (Amazon EC2). Both tools calculate the full distributions of response times in Continuous Time Markov Chains (CTMCs) but require a different amount of interprocessor communication, and hence scale different in Cloud Computing environments.

## 2 Elastic Capacity Management

In this section we present results of our case study on self-adaptive resource management in virtualized environments [23]. To avoid violations of service-level agreements (SLAs) or inefficient resource usage, capacity management has to be adopted continuously during system operation. For example, in Cloud Computing scenarios resources allocated to services need to be increased or decreased to reflect changes in application workloads. This is an approach on elastic capacity management based on online architecture-level performance models [25]. The goal is to maintain performance and efficient resource usage during run-time. In our evaluation we use the new SPECjEnterprise2010 benchmark[5].

### 2.1 Self-adaptive Resource Management

Our self-adaptive resource management follows the control loop model [11] which consists of four phases: *collect*, *analyse*, *decide* and *act*. For the *collect* phase, we assume that changes of the application workload are either announced by the customers (e.g., for an upcoming sales promotion) or by techniques like workload forecasting [5]. We then use the Palladio Component Model [3] and its performance prediction techniques to *analyze* the impact of these changes and to *decide* which actions to take. In this case study, the *act* phase covers the reconfiguration operations adding/removing application server cluster nodes and increasing/decreasing the number of virtual CPUs of a cluster node's virtual machine.

**Resource Allocation Algorithm** The following algorithm is executed if SLAs are violated or resources are used inefficiently. The goal is to find a new system

[5] SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjEnterprise2010 is located at `http://www.spec.org/jEnterprise2010`.

configuration which again maintains performance and resource efficiency. The algorithm is specified in generic terms, such that it can be applied to different types of resources and resource allocation operations. In short, the algorithm works on a set of services, resource types and SLAs. The SLAs specify, e.g., the requested average response time for a service at a given arrival rate. Each time there is a change of a specified SLA (e.g., a new client workload is scheduled for execution or a change in the workload intensity of an existing workload is forecast), we use our architecture-level performance models to predict the effect of this change on all SLAs. The algorithm can be divided into two phases: PUSH phase and PULL phase. If an SLA violation is detected, the PUSH phase of our algorithm is executed which allocates additional resources (PUSH additional resources into the system) until all client SLAs are satisfied. After the PUSH phase finishes, the PULL phase is executed to optimize the resource efficiency. If no SLAs are violated, the PULL phase starts directly to reduce the amount of used resources (PULL them out of the system).

*PUSH:* Basically, while there exists a client response time SLA that is violated, in this phase the algorithm increases the amount of allocated resources for all resource types used by the service which are overutilized. Increasing the number of allocated resources works as follows: If a there is an instance of an overutilized resource type (e.g., a VM) which has some processing resources available that are not allocated yet, additional resources are allocated (e.g., virtual CPUs). Otherwise, a new instance of this resource type is added (e.g., a new VM is started).

*PULL:* The PULL phase aims to optimize the resource efficiency by releasing resources that are not utilized efficiently under the current client workloads. In our algorithm, inefficient usage means the delta of maximum utilization and current utilization of a resource type is greater than a predefined constant, e.g., 20%. While there is a resource type assigned to service of the currently considered workload which is used inefficiently, the amount of resources allocated to this service will be decreased, i.e., for a resource type instance the capacity (e.g., virtual CPUs) is reduced. If the client SLAs are predicted to be violated after this change, the change is reversed.

### 2.2 Evaluation

In this section we briefly explain the SPECjEnterprise2010 benchmark and the experimental environment we used to evaluate our approach. Finally, we present the experimental results.

**SPECjEnterprise2010 Benchmark** We selected the SPECjEnterprise2010 benchmark application as a basis for our case study because it models a representative, state-of-the-art enterprise system. SPECjEnterprise2010 is a benchmark developed by SPEC's Java subcommittee to measure the end-to-end performance

and scalability of Java EE-based application servers. The benchmark workload is generated by an application that is modeled after an automobile manufacturer. As business scenarios, the application comprises customer relationship management (CRM), manufacturing and supply chain management (SCM).

The benchmark driver executes five benchmark operations. A dealer may *browse* through the catalog of cars, *purchase* cars or *manage* his dealership inventory, i.e., sell cars or cancel orders. A manufacturer may place *work orders* for manufacturing vehicles, either triggered per WebService or RMI call. In our experiments these benchmark operations function as the different services. To control the request arrival rate of each service individually, we had to slightly modify the benchmark driver. We split up the two driver domains and three manufacturing domains into five different domains, each invoking its own service. The resulting five independent services are called *Purchase*, *Manage*, *Browse*, *CreateVehicleEJB* and *CreateVehicleWS*.

**Architecture-level Performance Model** To make decisions in our control loop, we use a PCM model [3] as architecture-level performance model to predict the service response times and resource utilizations of the SPECjEnterprise2010 application for a specific load. The PCM model is semi-automatically extracted from a running benchmark application instance. As extraction method, we use the method in [7]. However, for this case study we extracted the entire benchmark application, i.e., including supplier domain, dealer domain, web tier and the asynchronous communication between the three domains. For reasons of brevity, the reader is referred to [23] for a detailed description of the PCM model instance.

**Experimental Setup** As hardware environment for the experiments, we use six blade servers from a cluster environment. Each server is equipped with two Intel Xeon E5430 4-core CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. On top of each machine, we run Citrix XenServer 5.5 as the virtualization layer. Inside the XenServer's VMs, we run the benchmark components. Each component runs in its own VM, initially equipped with 2 virtual CPUs (VCPUs). As operating system, these VMs execute CentOS 5.3. As Java EE application server, we use the Oracle Weblogic Server (WLS) 10.3.3. The load balancer is haproxy 1.4.8 using round-robin as load balancing strategy. The database is an Oracle 11g database server instance deployed on a VM with eight VCPUs on a separate node on Windows Server 2008. The SPECjEnterprise2010 benchmark application is deployed in a cluster of WLS nodes. For the evaluation, we considered reconfiguration options concerning the WLS cluster and the VCPUs the VMs are equipped with: WLS nodes are added to or removed from the WLS cluster, VCPUs are added to or removed from a WLS node's VM. These reconfigurations are applicable at run-time, i.e., can be applied while the benchmark application is running.

**Results** In the following section we present experimental results of our approach. First, we demonstrate how the approach behaves when the system work-

load increases. Next, we give an example how this approach can be used for elastic capacity management and show its benefits.

*Workload Growth:* In this scenario, we evaluate our approach when increasing the workload of all services deployed in our environment. We increase the load in two steps from 2x to 4x and 4x to 6x (see Figure 1). The standard workload (1x) is defined as request arrival rate (requests/second) for each service: (CreateVehicleEJB, 15), (CreateVehicleWS, 15), (Purchase, 12.5), (Manage, 12.5) and (Browse, 25). Our starting point is that five services are running on one node with three VCPUs ($c_1$) with 2x the standard workload and the following SLAs (CreateVehicleEJB, 30, 74ms), (CreateVehicleWS, 30, 74ms), (Purchase, 25, 130ms), (Manage, 25, 130ms), (Browse, 50, 130ms) which are initially satisfied. Now, we increase the workload to 4x the standard load. For this new workload, the reallocation algorithm detects a violation of the SLAs and recommends to reallocate the system resources using two nodes, one with four VCPUs and one using three VCPUs ($c_2$). Applying this configuration to our benchmark, the SLAs are satisfied. For the measurement results see Figure 1 a).

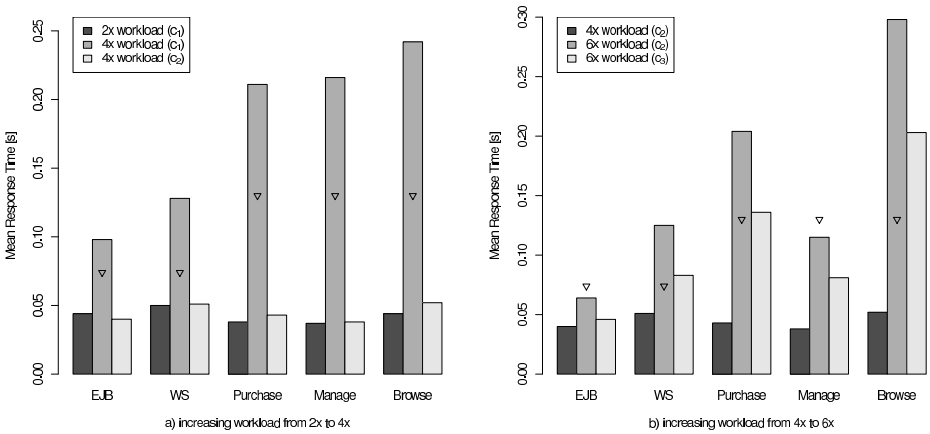



**Figure 1.** The response times when changing workload from 2x to 4x and 4x to 6x, respectively (SLAs denoted by $\triangledown$). The three bars depict the response times for all five services before the load increase, after the load increase, and after system reconfiguration.

In the second step, we increase the workload to 6x the standard load and do not change the SLAs. Again, this leads to a violation of the SLAs in our simulation results. Therefore, we apply our algorithm, finding a new suitable configuration with three nodes, two with four VCPUs and one with three VCPUs ($c_3$). The experiment results are depicted Figure 1 b). However, the results show that after reallocation the SLA of the Browse service is still slightly violated. This is not due to inaccuracy of our model, but rather due to scalability problems of

the database machine, which is not powerful enough to handle the new workload while satisfying the original SLAs. Hence, we are confident that given a more powerful database, the SLAs would be satisfied. The way this problem would be addressed in practice would be to either scale the database or renegotiate the SLAs. As both solutions can be handled with our online performance prediction mechanism, we plan to extend our approach with this solution in the future.

*Resource Usage and Efficiency:* After evaluating the functionality of our approach, this section discusses its benefits. Imagine a workload distribution over seven days like the one depicted in Figure 2. In a static scenario, one would assign three dedicated servers to guarantee the SLAs for the peak load. However, with our approach one can dynamically assign the system resources. In the static scenario, one would use three servers for seven days, whereas our approach needs only $1+2+3+2+3+1+1 = 13$ server days. Hence, in such a scenario, only 62% of the resources of the static assignment are needed and thereby almost 40% of the resources available can be saved.

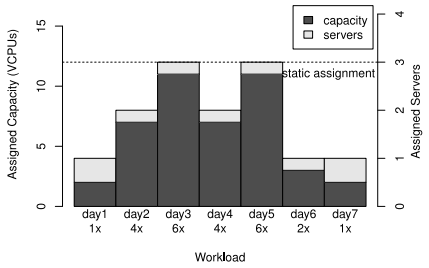



**Figure 2.** Assigned capacity and servers for a workload distribution over seven days.

### 2.3 Conclusions

This case study on self-adaptive resource management demonstrates that architecture-level performance models in combination with resource allocation algorithms can be applied to react to changes during runtime. It shows that it is possible to achieve elastic capacity management while satisfying specified SLAs. Exemplary, we showed how the system reacts on changes in the workload and how such an approach can save up to 40% of the resources. Also important to note is that this case study demonstrates that architecture-level performance models can be used effectively at runtime to support self-adaptiveness.

## 3 Case Study: Balancing Performance and Dependability Tradeoffs

Availability and responsiveness are crucial, although often conflicting, requirements for the multitier applications that implement critical business functional-

ity for many enterprises. Ensuring high availability requires the applications to be deployed with sufficient redundancy, potentially spanning several data centers. Today, large geographically dispersed hosting facilities provided by leading cloud computing providers have made wide area deployments practical for even low to medium scale applications. However, with such dispersion come consistency and synchronization overheads, and applications must often pay a performance penalty as a result. In traditional static deployments, application designers often tune such availability and performance tradeoffs manually after taking into consideration application architectures, workloads, and requirements.

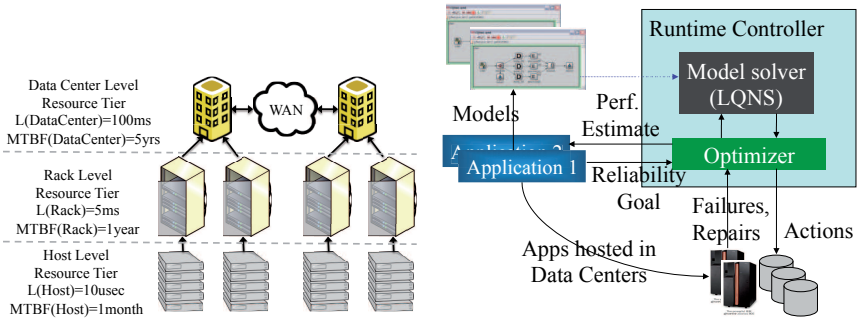
However, shared infrastructures such as compute clouds necessitate a rethinking of static deployment schemes. For example, resource contention might require relocation of applications to another machine rack, cluster, or even another data center. Additionally, current trends in system and data center design emphasize the use of large numbers of machines running cheap, less reliable commodity components that can fail often. 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 [13]. At the same time, skilled manpower is quickly becoming the most expensive resource, thus encouraging data center operators to batch repairs and replacement, thus increasing MTTR. in the process. In fact, portable "data-center in a box" designs (e.g., [21]) 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 run in increasingly dynamic environments in which parts of the infrastructure are in a failed state and static solutions to availability and performance tradeoffs will no longer suffice. However, dynamic solutions that redeploy multitier applications are challenging because they must not only balance availability against performance, but they must also factor in resource allocation between competing applications. Poor placement of a critical resource such as a database server may cause it to be a bottleneck for the whole application and as a result, the hosts where other tiers of the application are placed may become underutilized.

In this study, we show how online performance and availability models can be used to address these challenges and drive dynamic multitier application redeployment in the event of failures so as to minimize performance degradation while maintaining availability constraints. We build an online controller based on the models that regenerates the affected software components across clusters or data centers in the event of infrastructure failures, and reconfigures the entire system to run optimally on the remaining resources. Using simulation and fault injection studies, we show that this approach can provide high availability with far fewer resources than traditional approaches.

### 3.1 Performance and Availability Models

We consider a consolidated data-center environment in which a set of multitier applications $A$ are deployed on a set of physical hosts $H$ located in a number of data centers. The hosts are organized into racks, clusters, and data centers in a




(a) Resource Levels Example (b) Approach overview

**Figure 3.** Resource levels example and approach overview.

resource hierarchy $(R, \leq_R)$, where $R$ is the set of "resource groups" (i.e., machine, rack, cluster, data-center) and $\leq_R$ specifies a direct hosting relationship between the groups, e.g., Host1 $\leq_R$ Rack1 $\leq_R$ DataCenter1. The hosting relation $\leq_R^*$ is the transitive closure of $\leq_R$, e.g., Host1 $\leq_R^*$ DataCenter1 indicates that Host1 is directly or indirectly hosted in DataCenter1. Figure 3(a) shows an example resource hierarchy with 20 machines distributed across four racks in two data centers. Two resource groups 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.

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 group $r$ by $\text{MTBF}_r$. 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 to avoid single points of failure. Each application $a$ may support multiple transaction types $T_a$. For example, the RUBiS [10] 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 application's workload $w_a$ is given by a vector of request rates $w_a^t$ for its transactions. Each application-component replica executes in its own virtual machine (VM) [2] on a physical host anywhere in the resource hierarchy that it can share with other VMs. Each VM is allocated a share of the host's CPU capacity that is enforced by Xen's credit-based scheduler.

*Availability Models.* We consider an application to be available when at least one replica of each component is running on an operational machine, and define

availability as the fraction of time the application is available. A replication level of at least two for each of the application's component types is necessary to avoid single points of failure, but 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 causes application failure. Therefore, we allow each application to specify its desired availability and use information about the system's recovery policy codified by the MTTR to calculate the application's minimum desired "mean time between failures", or $\text{MTBF}_a$ as: $\text{MTBF}_\text{a} \geq \frac{\text{Availability}_\text{a} \cdot \text{MTTR}}{1 - \text{Availability}_\text{a}}$.

We can now calculate the application's actual MTBF for a given placement of its components across the resource hierarchy. Assume that each resource group $r$ fails independently according to a Poisson failure process with rate $\lambda_r = 1/MTBF_r$ and each failure disables all the application components the group contains. If the replication level of any application component type drops to zero as a result of a resource failure, then the application fails. For each of application $a$'s component types $n_a$, let $r^{max}(n_a)$ be the highest level resource group such that all replicas of $n_a$ are contained in that resource group. E.g., if an application's database had 2 replicas hosted in DataCenter1:Rack1:Host1 and DataCenter1:Rack2:Host3, then $r^{max}(db_a) = \text{DataCenter1}$. Only failures at resource levels $r^{max}(n_a)$ or higher will cause an application failure by causing the replication level of the component type $n_a$ to fall to zero.

Under these assumptions, the overall failure arrival process is also Poisson with rate $\sum_{r \in R} \lambda_r$. A failure event affects resource group $r$ 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., $r^{max}(n_a) \leq_R^* r$ (Condition 1). Since this condition only filter resource groups, the application failure process is also Poisson with a rate equal to the sum of $\lambda_r$ over resource groups for which condition 1 is true. $r^{max}(n_a)$ depends only the exact system configuration, so the application failure process has a constant rate until the system is reconfigured by the controller. Thus, the MTBF for application $a$ in a system configuration $c$ is given by:

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

This equation assumes that no additional failures occur in the time window between the first failure and the time the controller finishes reconfiguring the system. While this is not strictly true, it is a reasonable assumption because the reconfiguration actions (VM instantiation, migration, CPU capacity changes) are very short compared to typical resource MTBF values.

*Performance Models.* To quantify the performance of alternative system configurations, we construct application models using the layered queuing network modeling formalism [31] to predict the response times of application transactions and the corresponding resource utilization demands for each replica for a given workload and system configuration (i.e., the CPU capacity assigned to each application VM). Each application component is modeled as a FCFS queue,

while hardware resources (e.g., CPU and disk) are modeled as processor sharing (PS) queues. Interactions between tiers triggered by a transaction are modeled as synchronous calls in the queuing network, and our models also account for the resource sharing overhead imposed by Xen. The parameters for models (e.g., per-transaction service time at each queue) are measured in an offline measurement phase, where each application is instrumented using system call interception. Then, delays between incoming and outgoing messages are measured per transaction. Details of the LQN models and their validation can be found in [24].

We compute the application's mean response time in a new configuration as the sum of the response time $RT(a,t)$ of each transaction $t$ weighted by the fraction $\gamma(a,t)$ of the transaction in the application's workload mix. The response time degradation in a potential new configuration is simply the difference between the predicted mean response time in the new configuration and the mean response time in the original configuration before the failure.

### 3.2 Online Optimization Algorithm

Our approach, as shown in Figure 3(b), is realized by a runtime controller that monitors the system and which, when a failure or recovery occurs, reconfigures all applications. To do so, it uses standard virtual machine techniques - it can either migrate each application component's VM to another host, or change the CPU share allocated to the VM on its current host. The controller chooses actions that minimize the mean performance degradation (across all applications) as a result of the failure while still maintaining the desired level of replication and application MTBF (Equation 1). 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. Additionally, applications requiring high levels of reliability will have to be distributed across higher resource levels to prevent single failures from impacting multiple replicas.

The optimization is carried out over the large 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 $c.\text{host}(n_k)$, and (b) the CPU share cap $c.\text{cap}(n_k)$. The CPU cap $c.\text{cap}(n_k)$ allocated to a replica impacts it's processing speed, and thus the application's end-to-end performance. However, for a fixed CPU cap, the choice of machine on which to host a component only depends on the network latency of the machine to the locations of the other application components. Furthermore, according to our definition of $L(rl)$ in Section 3.1, the network latency is a function of the resource level rather than individual resources. For example, a resource level of "rack" would require placement of replicas of the same type across different hosts in the same rack, while a resource level of "whole system" would entail placing the replicas on hosts in different data centers.

The optimization algorithm determines the values of the parameters that affect application performance (CPU cap, resource level) by using a gradient descent search to minimize performance degradation. The algorithm starts with

the maximum value of CPU cap (i.e., 1.0) for every replica and the lowest permissible resource level for each tier such that the application's MTBF given by equation 1 is higher than the application's desired MTBF. Once an initial value for the parameters is chosen for every replica, the algorithm attempts to fit the replicas into the available resources using a bin-packing algorithm that respects each replica's choice of resource level and uses the CPU cap as the "volume" of the replica. If a fit cannot be found, the algorithm executes an additional iteration of the gradient descent to either choose to lower the value of the CPU cap of a single replica to reduce CPU requirements, or to increase the resource levels of a single application tier to increase the flexibility the bin-packer has when distributing replicas. The option (which application, which tier, and whether to reduce the CPU cap or increase the resource level) that results in the least amount of performance degradation is chosen. The LQNS queuing model described above is used to estimate the performance degradation in the new configuration. The bin-packing is attempted again and the process repeats until a successful fit in the available resources is found. More details of the optimization algorithm can be found in.

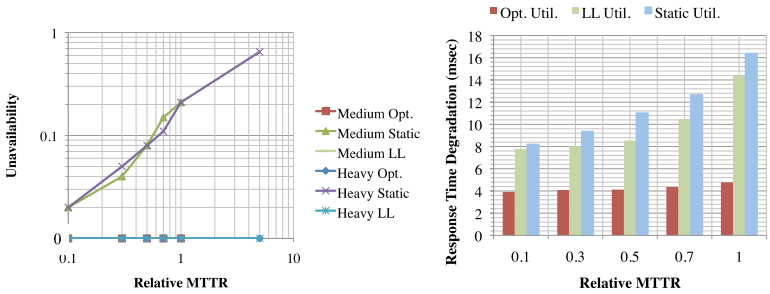
Upon finding a successful fit, the optimizer calculates the difference between the original configuration and new configuration for each replica, and returns the set of actions (migrate VM, adjust CPU cap, re-instantiate VM) needed to affect the change. The durations of these actions are relatively short 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 [12].

### 3.3 Simulation based Evaluation

In this section, we present simulation results using a simulator written in the Java based SSJ framework [26]. The target application for the experiments is the RUBiS online auction benchmark. We created the LQNS model using offline measurements from [24] 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 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 host within the same level of the resource hierarchy. 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 simulate the three strategies in a cloud setup consisting of two data centers with three clusters each, 3 racks in each cluster, and 4 machines in each rack. The communication delays between two machines in a rack are $D$,




(a) Unavailability (b) Performance Degradation

**Figure 4.** Simulation results.

while those between machines in different racks, clusters, and data centers are $1.5D$, $2D$, and $2.5D$, respectively. The setup hosts 6 instances of RUBiS: 3 gold instances each with a weight of 5 (Equation 1) and 3 silver instances with a weight of 1 (Equation 1). For all instances, each of the three tiers is replicated twice. The gold instances offer higher availability and require tiers to be replicated at-least across separate clusters, while the silver application tiers can be replicated across racks in the same cluster. The workload is set to 30 and 60 requests/sec for the gold and silver instances, respectively. Each VM is initially allocated 80% of one physical CPU.

For each strategy, we run fault injection experiments in which failures are simulated at different levels of the hierarchy (i.e., data center, cluster, rack, host) using a Poisson process 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. To make the results applicable for systems with different MTBFs and MTTRs, we report all times normalized to the host-level MTBF $M_f$, which was set to 1.0, For repair, we vary the per host relative MTTR from 0.1 to 1, indicating that repair takes from 10% to 100% of the MTBF. Each simulation runs for a normalized time period of 10 (i.e., 10 failures per run on the average), and we repeat each experiment 10 times. For each experiment, we calculate both the availability of the system and the performance degradation.

Figure 4(a) shows the unavailability of the system as a function of the relative MTTR. Both the Opt and LL strategies achieve 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 very short compared to typical MTBF values in practice.

Figure 4(b) shows the mean performance degradation $D$ of the applications computed over the period that they are available vs. the MTTR. 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, it may be better to do nothing (i.e., Static) than reallocating those VMs to machines that are running Gold instances (which have a higher impact on the weighted mean response time) and slowing them down. LL also cannot determine which components are bottlenecks and often makes decisions based on small differences in host CPU utilizations (since all of them are high). It can end up co-locating a regenerated VM with a bottleneck resource, thereby greatly degrading the response time. On the other hand, the Opt strategy can avoid these bottlenecks using its queuing model, and performs significantly better than both Static and LL by exhibiting little performance degradation even at high relative MTTR values.

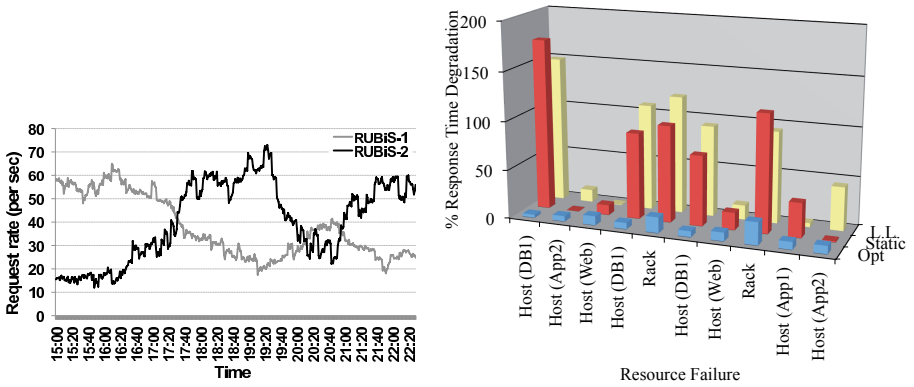
### 3.4 Fault Injection based Evaluation

Next, we experimentally evaluate the Opt, Static, and LL strategies using fault injection experiments on a system subjected to actual failures and a realistic workload. Our testbed contains 10 machines divided into two racks of 5 each. Each host has an Intel Pentium 4 1.80GHz processor, 1 GB RAM, and a 100 Mb Ethernet interface and runs Linux kernel 2.6.18 guest OS VMs on an open-source Xen version 3.2.0 hypervisor. The controller is run on a separate server.

The hosted applications are two instances of the 3-tier servlet version of RUBiS running on an Apache 2.0.54 webserver, a Tomcat 5.0.28 application server, and MySQL 3.23.58 database. Each tier has two replicas, and each replica runs in its own VM for a total of 12 VMs. Replicas for the same tier are constrained to run in different racks. In the initial configuration, the VMs hosting the Tomcat and MySQL replicas are allocated 80% of a physical CPU capacity and the VMs hosting Apache replicas are allocated 40% of a physical CPU capacity. 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.

The applications are subjected to the workloads shown in Figure 5(a). These traces are produced by a user emulator that simulates actual users using a semi-Markov model. Each state of semi-Markov model corresponds to a single transaction. Transitions between states $s' \rightarrow s''$ encode the probability that a user issues the transaction destination transaction $s''$ after visiting the source page $s'$. The user is assumed to spend a normally distributed amount of random "think time" between every consecutive transaction. The number of concurrent users at any given time varies, and we obtain these variations from actual user traces from publicly available web-site logs [1, 14].

Both individual host and rack failures (i.e., a correlated failure of all machines in a rack due to a common cause such as power supply or switch/router) are injected. In each run, a single failure is injected at a random time instant




(a) Workload

(b) Response time degradation (%) after failure and reconfiguration

**Figure 5.** Cloud simulation results.

and the mean response time of the applications before and after the injection and reconfiguration is measured to calculate the performance degradation. Each strategy is subjected to failures at the same time instant and workload and the mean performance degradation across all transactions is reported. The results for one of the RUBiS instances are shown in Figure 5(b). Across all failures, the average performance degradation for the Static and LL strategies is 46% and 47%, respectively, while it is only 9.5% for the Opt. controller. The gap between Static and LL is small because the initial configuration has no single point of failure, and the relatively light workload allows the Static approach to operate after a single failure without requiring VM regeneration. The large differences between Opt. and LL demonstrate the benefits of taking performance bottlenecks into account during reconfiguration. The Opt. controller has very low degradation even when entire racks fail.

### 3.5 Conclusion

In this study, we have examined how online controllers can be constructed to optimize multitier application placements by balancing performance and availability tradeoffs. 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 performance and availability models to minimize the resulting performance degradation. Experimental results show that the proposed approach provides improved performance than classical approaches.

## 4 Computational and Communication Scalability of EC2

Stochastic models of real-life computer and communication systems allow engineers to analyse the correctness and performance of such systems at design time. This allows for problems to be detected and choices to be investigated much more quickly and cheaply than if such investigations are delayed until after the system has been implemented. Markov chains are one of the most commonly-encountered modelling formalisms, but to capture even the most essential behaviour of a real-life system may require a Markov chain with many millions of states. The analysis of such chains will require the combined compute power and memory capacity of a number of computers in parallel; for example, see [4,6,8,18,20,22,27,30]. Typical quantities of interest are long-run or *steady-state* probability distributions and the distributions of response times between specified initial and goal states.

To exploit the power of these implementations, the user is typically required to possess a dedicated computational cluster or network of workstations. Such hardware is, however, expensive to buy and to run, requires sufficient space with associated power and cooling to house it, and staff to maintain it. With the ever-present pressure on academic research budgets, it is conceivable that individual research groups will struggle to continue to acquire such resources for themselves. Cloud computing holds the promise of dramatically reducing these overheads.

A key concern in using existing performance analysis tools in the cloud is how well those tools themselves perform in this environment, as their performance in this shared environment could well differ from their performance on dedicated hardware. Cloud computing offers the ability to make use of large numbers of processors far more cheaply than we could ourselves own, but if our tools cannot efficiently use these extra resources then they will need either to be modified or to be replaced with tools that can.

In this section we study the scalability of two of our previously-presented performance analysis tools: a Laplace transform-based response time analyser [16] and the HYpergraph-based Distributed Response-time Analyser (HYDRA) [17, 18]. Both tools calculate the full distributions of response times in Continuous Time Markov Chains (CTMCs), which can then be used to reason about a wide range of performance requirements in formal models of systems.

We compare the two tools' scalabilities in a cloud computing environment (Amazon EC2) and a variety of traditional environments in the context of a case study analysis of a CTMC model. We expect that the Laplace transform-based tool will scale well in all environments because of the minimal amount of interprocessor communication that it requires, but that HYDRA may suffer in environments with limited network bandwidth despite the fact that it employs a data partitioning scheme that minimises interprocessor communication.

### 4.1 Performance Analysis Tools

We will study the scalability of two previously-presented performance analysis tools: a Laplace transform inverter [16] and the HYpergraph-based Distributed

Response-time Analyser (HYDRA) [17,18]. Although the core computation carried out by both tools is repeated sparse matrix–vector multiplication, the way in which they parallelise the problem is different and consequently they place very different communication loads on the network.

**Laplace Transform Inverter** The distributed Laplace transform inverter is written in C++ and uses the Message Passing Interface (MPI) [19] standard, so it is portable to a wide variety of parallel computers and workstation clusters. It features a master-slave architecture that ensures a good load balance and very high utilisation of slave processors. In addition, there is no inter-slave communication and the amount of master-slave communication is low. We therefore expect this tool to exhibit good scalability.

**HYDRA** HYDRA is also implemented in C++ and uses MPI. The key opportunity for parallelism in HYDRA is in the repeated sparse matrix–vector multiplications that form the core of the implemented algorithm. To perform these operations efficiently in parallel it is necessary to map the non-zero matrix elements onto processors such that the computational load is balanced and communication between processors is minimised. To achieve this, we use hypergraph partitioning to assign matrix rows and corresponding vector elements to processors [9]. Our previous work has observed that this gives HYDRA good scalability on both parallel computers with fast interconnection networks and also on networks of workstations connected via switched Ethernet [17,18].

### 4.2 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (Amazon EC2) is a service that allows users both to purchase computing resources on-demand and also to reserve them to guarantee availability in the future. Central to EC2 are Amazon Machine Images (AMIs), which are instantiations of the Linux or Windows operating system that are brought into being by the user and run as virtual machines. Amazon have a range of standard AMIs, based on Windows and various versions of Linux, that come pre-installed with commonly-used packages as well as providing tools to enable users to build their own AMIs containing exactly the applications and packages that they require.

Both of our tools described in the previous section require MPI and, although none of the standard Amazon AMIs include this, there is a user-produced AMI that does [28,29]. This AMI costs $0.085 per instance per hour,[6] and is only available in the US-N. Virginia region of EC2. Similarly, although Amazon has recently released a dedicated Cluster Compute image (Cluster Compute Quadruple Extra Large, or `cc1.4xlarge`) with access to 10Gbps Ethernet interconnection, this image does not come with MPI installed as standard. By following the publicly-available instructions of [28], however, we were able to create our own

[6] See `http://aws.amazon.com/ec2/pricing/` for a full list of rates.

custom Cluster Compute image that included MPI. It costs $1.60 per instances per hour to run and is also only available in the US-N. Virginia region.

### 4.3 Results

| Name | Type | CPU | RAM | Network |
|---|---|---|---|---|
| PC (2004) | Workstation | Intel Pentium 4 2.0GHz | 512MB | 100Mbps Ethernet |
| PC (2010) | Workstation | Intel Core2 Duo 3.0GHz | 4GB | 1Gbps Ethernet |
| Camelot | Cluster | Opteron dual-core 2.2GHz | 8GB | 2.5Gbps Infiniband |
| Amazon | EC2 Small Instance | *c.* Opteron 1.0-1.2GHz | 1.7GB | Unknown |

**Table 1.** Summary of the four architectures on which the Laplace transform inversion tool was executed.

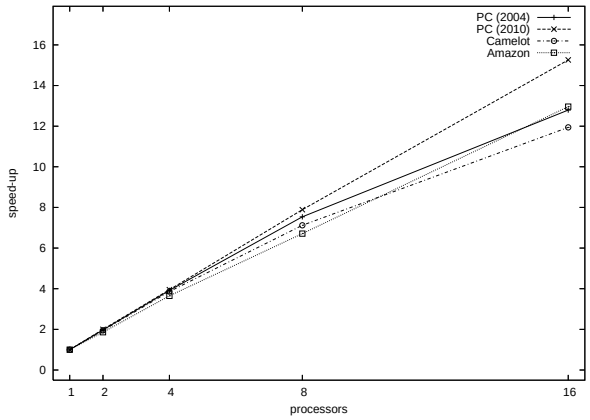
| | PC (2004) | | | PC (2010) | | | Camelot | | | Amazon | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ |
| 1 | 5 096.0 | 1.0 | 1.0 | 1 190.5 | 1.00 | 1.00 | 4 181.3 | 1.00 | 1.00 | 2 835.9 | 1.00 | 1.00 |
| 2 | 2 582.6 | 1.97 | 0.99 | 592.4 | 2.00 | 1.00 | 2 149.1 | 1.95 | 0.97 | 1 522.4 | 1.86 | 0.93 |
| 4 | 1 298.4 | 3.92 | 0.98 | 301.4 | 3.95 | 0.99 | 1 083.1 | 3.86 | 0.97 | 776.2 | 3.65 | 0.91 |
| 8 | 675.8 | 7.54 | 0.94 | 150.9 | 7.89 | 0.99 | 587.6 | 7.12 | 0.89 | 422.7 | 6.71 | 0.83 |
| 16 | 398.4 | 12.79 | 0.80 | 78.0 | 15.26 | 0.95 | 350.3 | 11.94 | 0.75 | 218.8 | 12.96 | 0.81 |

**Table 2.** Average run-times in seconds (T), speed-ups ($S_p$) and efficiencies ($E_p$) for $p$-processor response time density calculations in a 537 768 state model using the Laplace transform inversion tool.

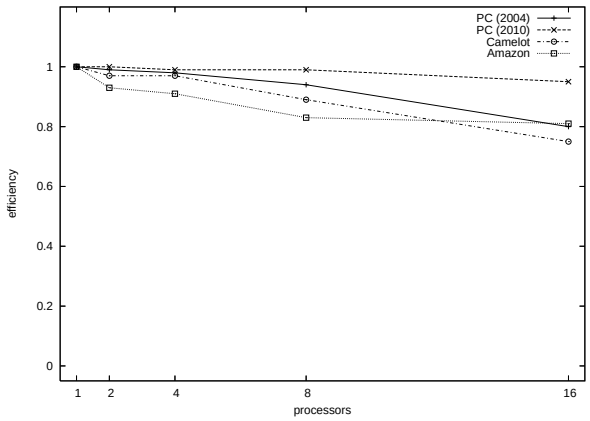
**Laplace Transform Inverter** These results are presented for four architectures and are reproduced from [15] to provide a basis for comparison with the new results in the next section. Tab. 1 summarises the processor speeds, main memory and network bandwidths of the four architectures. Tab. 2 shows the run-times, speed-ups and efficiencies for the calculation of response time densities on $p$ processors for a 537 768 state model. Corresponding graphs of speed-up and efficiency are shown in Fig. 6. Note that run-times are averages of 5 runs.

We expect the Laplace transform tool to exhibit good scalability as there is very little inter-processor communication, and this is shown to be the case. Indeed, it is noticeable that on EC2 the speed-up trend is almost linear, which suggests that the master-slave architecture with minimal intercommunication is an appropriate design for cloud-based parallel tools.

**HYDRA** These results are presented for five architectures, four of which are reproduced from [15]. Tab. 3 summarises the processor speeds, main memory




(a) Speed-up




(b) Efficiency

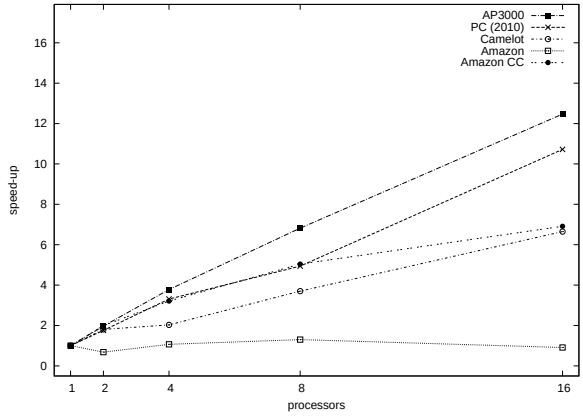
**Figure 6.** Speed-up and efficiency graphs for $p$-processor response time density calculations in a 537 768 state model using the Laplace transform inversion tool.

| Name | Type | CPU | RAM | Network |
|---|---|---|---|---|
| AP3000 | Distributed-memory parallel computer | UltraSparc 300MHz | 256MB | 520Mbps mesh |
| PC (2010) | Workstation | Intel Core2 Duo 3.0GHz | 4GB | 1Gbps Ethernet |
| Camelot | Cluster | Opteron dual-core 2.2GHz | 8GB | 2.5Gbps Infiniband |
| Amazon | EC2 Small Instance | *c.* Opteron 1.0-1.2GHz | 1.7GB | Unknown |
| Amazon CC | EC2 Compute Cluster | Xeon quad-core 2.93GHz | 23GB | 10Gbps Ethernet |

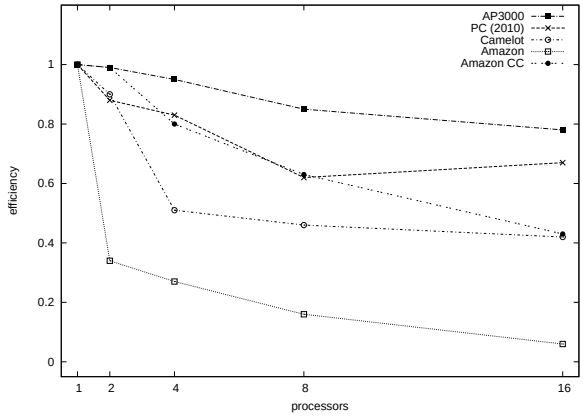
**Table 3.** Summary of the five architectures on which HYDRA was executed.

| | AP3000 | | | PC (2010) | | | Camelot | | | Amazon | | | Amazon CC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ | T | $S_p$ | $E_p$ |
| 1 | 1 243.3 | 1.00 | 1.00 | 76.8 | 1.00 | 1.00 | 178.1 | 1.00 | 1.00 | 112.5 | 1.00 | 1.00 | 61.77 | 1.00 | 1.00 |
| 2 | 630.5 | 1.97 | 0.99 | 43.5 | 1.76 | 0.88 | 98.7 | 1.81 | 0.90 | 166.2 | 0.68 | 0.34 | 31.05 | 1.99 | 0.99 |
| 4 | 328.2 | 3.78 | 0.95 | 23.2 | 3.31 | 0.83 | 87.9 | 2.03 | 0.51 | 104.8 | 1.07 | 0.27 | 19.25 | 3.21 | 0.80 |
| 8 | 182.3 | 6.82 | 0.85 | 15.5 | 4.94 | 0.62 | 48.2 | 3.70 | 0.46 | 86.3 | 1.30 | 0.16 | 12.26 | 5.04 | 0.63 |
| 16 | 99.7 | 12.47 | 0.78 | 7.2 | 10.72 | 0.67 | 26.8 | 6.65 | 0.42 | 123.4 | 0.91 | 0.06 | 8.94 | 6.91 | 0.43 |

**Table 4.** Average run-times in seconds (T), speed-ups ($S_p$) and efficiencies ($E_p$) for $p$-processor response time density calculations in a 1 639 440 state model using HYDRA.




(a) Speed-up




(b) Efficiency

**Figure 7.** Speed-up and efficiency graphs for $p$-processor response time density calculations in a 1 639 440 state model using HYDRA.

and network bandwidths of the five architectures. As parallel sparse matrix–vector multiplication potentially requires a great deal of data to be exchanged at each iteration of the solution, we also investigate HYDRA's scalability when executed on Amazon's Compute Cluster instances. In an effort to ensure that we the effect of the network is included in our results, we used two instances for all values of $p > 1$, with at least one process assigned to each instance.

Tab. 4 shows the run-times, speed-ups and efficiencies for the calculation of response time densities on $p$ processors for a 1 639 440 state model. Corresponding graphs of speed-up and efficiency are shown in Fig. 7. Once again, these run-times were averaged over 5 runs. Although the use of hypergraph partitioning minimises the amount of data that must be sent, we observe that the speed-ups achieved are accordingly lower than for the Laplace transformer inverter. We also observe that the scalability of HYDRA on the standard Amazon EC2 instances is the worst of all five architectures. Although we expected the speed-up and efficiency to be lower than on the dedicated hardware platforms, it is still very surprising to see just how badly HYDRA fares in the cloud.

| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Amazon | \$0.09 | \$0.17 | \$0.34 | \$0.68 | \$1.36 |
| Amazon CC | \$1.60 | \$3.20 | \$3.20 | \$3.20 | \$3.20 |

**Table 5.** Average costs (to the nearest whole cent) for HYDRA execution.

The interconnection of the Cluster Compute instances clearly provides far higher bandwidth than the network connecting standard EC2 instances, and as a result HYDRA's scalability on these AMIs is much more in line with that experienced in dedicated HPC environments. This higher performance comes at increased cost, however, as can be seen from Tab. 5. Note that Amazon charges by the hour and that each Cluster Compute image provides 8 CPUs (hence why only two are required to run 16 HYDRA processes).

### 4.4 Conclusion

We observed that the Laplace transform tool scaled much better in the cloud than HYDRA, but that HYDRA's scalability improved dramatically when it was executed on Amazon's new Cluster Compute instances. This suggests that there are now cloud computing services that can rival traditional dedicated HPC environments; it should be recalled, however, that these instances were significantly more expensive than the standard EC2 ones.

## 5 Acknowledgements

The work of Samuel Kounev, Fabian Brosig and Nikolaus Huber was funded by the German Research Foundation (DFG) under grant No. KO 3445/6-1. The

work of Nicholas Dingle was funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/I006702/1 "Novel Asynchronous Algorithms and Software for Large Sparse Systems".

## References

1. M. Arlitt and T. Jin. Workload charaterization of the 1998 world cup web site. In *HP Technical Report, HPL-99-35*, 1999.
2. 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. of 19th Symp. on Operating Sys. Principles*, pages 164–177, 2003.
3. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 2009.
4. M. Benzi and M. Tuma. A parallel solver for large-scale Markov chains. *Applied Numerical Mathematics*, 41:135–153, 2002.
5. G. Box and G. Jenkins. *Time series analysis: forecasting and control*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1994.
6. J. Bradley, N. Dingle, W. Knottenbelt, and H. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. *Linear Algebra and its Applications*, 386:311–334, 2004.
7. F. Brosig, S. Kounev, and K. Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of ROSSA 2009*. ACM, Oct. 2009.
8. P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In *Proceedings of the 3rd International Conference on the Numerical Solution of Markov Chains (NSMC'99)*, pages 76–95, Zaragoza, Spain, September 1999.
9. U. Catalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.
10. E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proc. of the 4th ACM/IFIP/USENIX Intl. Middleware Conf.*, Rio de Janeiro, Brazil, 2003.
11. B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, 2009.
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 Sym. on Networked Sys. Design and Implementation*, 2005.
13. 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.
14. J. Dilley. Web server workload charaterization. In *HP Technical Report, HPL-96-160*, 1996.
15. N. Dingle. An empirical study of the scalability of performance analysis tools in the cloud. In *Proc. 26th UK Performance Engineering Workshop (UKPEW'10)*, pages 9–16, Warwick, UK, July 2010.

16. N. Dingle, P. Harrison, and W. Knottenbelt. Response time densities in Generalised Stochastic Petri Net models. In *Proc. 3rd International Workshop on Software and Performance (WOSP'02)*, pages 46–54, Rome, July 24th–26th 2002.
17. N. Dingle, P. Harrison, and W. Knottenbelt. HYDRA: HYpergraph-based Distributed Response-time Analyser. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 215–219, Las Vegas NV, USA, June 23rd–26th 2003.
18. N. Dingle, P. Harrison, and W. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.
19. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge MA, 1994.
20. M. Grottke, V. Apte, K. Trivedi, and S. Woolet. Response time distributions in networks of queues. In R. Boucherie and N. Dijk, editors, *Queueing Networks*, volume 154 of *International Series in Operations Research and Management Science*, pages 587–641. Springer US, 2011.
21. J. R. Hamilton. An architecture for modular data centers. In *Proc. of the Conf. on Innovative Data Sys. Research*, pages 306–313, 2007.
22. P. Harrison and W. Knottenbelt. Passage time distributions in large Markov chains. In *Proceedings of ACM SIGMETRICS 2002*, pages 77–85, Marina Del Rey CA, June 2002.
23. N. Huber, F. Brosig, and S. Kounev. Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *SEAMS'11: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011.
24. G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments,. In *Proc. of the 5th IEEE Intl. Conf. on Autonomic Computing*, pages 23–32, June 2008.
25. H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, August 2009.
26. P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: a framework for stochastic simulation in Java. In *Proc. of the Winter Simul. Conf.*, pages 234–242, 2002.
27. A. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, May 1993.
28. P. Skomoroch. MPI cluster programming with Python and Amazon EC2. In *Proc. 6th Annual Python Community Conference (PyCon'08)*, Chicago, March 2008. `http://www.datawrangling.com/mpi-cluster-with-python-and-amazon-ec2-part-2-of-3`.
29. P. Skomoroch. Data Wrangling Image: Fedora Core 6 MPI Compute Node with Python Libraries, 2010. AMI ID: ami-3e836657, `http://developer.amazonwebservices.com/connect/entry.jspa?categoryID=101&externalID=705`.
30. K. Trivedi, S. Ramani, and R. Fricks. Recent advances in modeling response-time distributions in real-time systems. *Proceedings of the IEEE*, 91(7):1023–1037, July 2003.
31. 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.