Autonomous Resource Provisioning for Multi-Service Web Applications*

Jiang Dejun
Vrije Universiteit
Amsterdam, The Netherlands
jiangdj@few.vu.nl

Guillaume Pierre
Vrije Universiteit
Amsterdam, The Netherlands
gpierre@cs.vu.nl

Chi-Hung Chi
Tsinghua University
Beijing, China
chichihung@mail.tsinghua.edu.cn

ABSTRACT

Dynamic resource provisioning aims at maintaining the end-to-end response time of a web application within a predefined SLA. Although the topic has been well studied for monolithic applications, provisioning resources for applications composed of multiple services remains a challenge. When the SLA is violated, one must decide *which* service(s) should be reprovisioned for optimal effect. We propose to assign an SLA only to the front-end service. Other services are not given any particular response time objectives. Services are autonomously responsible for their own provisioning operations and collaboratively negotiate performance objectives with each other to decide the provisioning service(s). We demonstrate through extensive experiments that our system can add/remove/shift both servers and caches within an entire multi-service application under varying workloads to meet the SLA target and improve resource utilization.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of systems**]: Design studies; H.3.4 [**Information Storage and Retrieval**]: Systems and Software.

General Terms

Performance.

Keywords

Resource provisioning, Multi-service application

1. INTRODUCTION

Major web sites such as Amazon.com and eBay are not designed as monolithic 3-tier applications but as a complex group of independent services querying each other [5, 11]. A service is a self-contained application providing elementary functionality, such as a database holding customer information or an application serving search requests. Web pages delivered to the users are generated by composing the results of many such services based on pre-defined workflows [11]. Services hide their internal implementation details from the outside world and expose functionality through standard invocation interfaces. Services participating in an application are typically composed in a directed acyclic graph.

To provide acceptable performance to their customers, application providers often impose themselves a Service Level Agreement (SLA) which defines for example the maximum average response time that the application should offer. One can then apply dynamic resource provisioning to respect the SLA target by adding resources when necessary to maintain the response time objective, and removing resources when possible without violating the SLA.

An essential question in resource provisioning of multi-service web applications is to select *which* service(s) should be (de-)provisioned such that the whole application maintains acceptable performance at minimal cost. This is a challenge because multi-service applications involve large number of components that have complex relationships with each other. For example, adding a cache to one service does not only improve its own response time, but also causes less traffic to the backend services it invokes.

One possible approach models the entire application as a single queuing network. However, when dealing with multi-service applications, such a model can become extremely complex to capture all services relationships and techniques such as caching. Another approach assigns a fixed SLA to each service separately. The SLA of the front-end service is trivially defined as the response time objective of the whole application. However, we show in Section 4.3.2 that no choice of internal service SLAs can match the performance of our system: the per-service SLA approach necessarily wastes resources as it makes certain services struggle to maintain their SLAs when an equivalent gain of end-to-end performance could be gained easier by reprovisioning another service.

We claim that only the front-end service should be given an SLA. A user commonly does not care about the performance of each particular service involved in the application, but only in the end-to-end response time that she observes. On the other hand, no service other than the front-end should have a specific SLA. Instead, each service should be autonomously responsible for its own provisioning by collaboratively negotiating its performance objectives with the other services to maintain the front-end's response time within the SLA. Negotiation between services is based on "what-if analysis:" each service continuously estimates the performance it would have in case it was assigned more/less resources, or if it received more/less traffic. The front-end

*This work is partially supported by the 863 HighTech Program of China under award #2008AA01Z12.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

service finally selects the optimal service(s) for resource provisioning from the perspective of the whole application.

We show that our system allows one to effectively provision resources to both traditional multi-tier web applications and complex multi-service applications. To our best knowledge, no other published algorithm addresses complex service invocation graphs. Our scheme also supports the provisioning of cache instances. Furthermore, in the case of subtle changes in workload patterns, a previous provisioning decision may need to be revoked without adding or removing resources, but by reassigning resources from one service to another. Our system allows such resource reorganization so as to accommodate long-term changes in user behavior.

This paper is organized as follows: Section 2 introduces the related work. Then, Section 3 presents our resource provisioning system, and Section 4 evaluates its performance for both multi-tier and multi-service web applications. Finally, Section 5 concludes.

2. RELATED WORK

Many research efforts address resource provisioning for single-tier [1, 4] or multi-tier Web applications [7, 10, 12, 14, 15, 17]. Some of them only model the most constrained tier of the web application [10, 17], or considerably simplify the operation model of each tier [7]. Others model the interactions across tiers and thus address the bottleneck shifts across tiers [12, 14, 15]. These models capture the performance impacts of techniques such as caching and database replication. In addition, Urgaonkar *et al.* handle session-based workload and concurrency limits at different tiers [14]. These works inspire our own performance model. However, they all assume that web applications consist of one or more tiers organized in a single line. We focus here on multi-service web applications constructed as directed acyclic graphs, which is largely different from these works.

Few works address provisioning in multi-service applications. Wu *et al.* model workflow patterns within multi-service applications to predict future workloads of each service component [18]. One can derive the number of required servers per service. However, this model assumes that each server has a fixed maximum capacity, which we consider as largely equivalent to assigning an SLA to each service.

A related topic is to decide when resources should be provisioned [15, 16]. The issue there is that provisioning resources takes time so advance planning is necessary. However, as virtualization is increasingly applied to provision web applications, allocating new resources becomes much faster: on-the-fly cloning of hundreds of virtual machine can happen within sub-second [9]. In contrast, we focus on selecting *which* service(s) to provision rather than when.

Finally, Almeida *et al.* aim to determine short-term resource demands and long-term capacity requirements for multiple applications sharing a common set of resources [2]. However, this problem is different from ours: the problem there is to arbitrate between multiple disjoint applications competing for the same resources. We take a different approach where resources are assumed to be infinitely available, as is typically the case in data centers or clouds. Here, the issue is to maintain acceptable performance of individual multi-service applications at minimum cost.

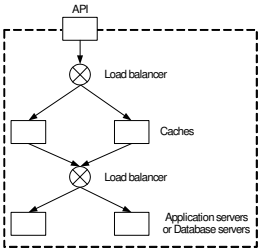


Figure 1: Hosting architecture of a single service

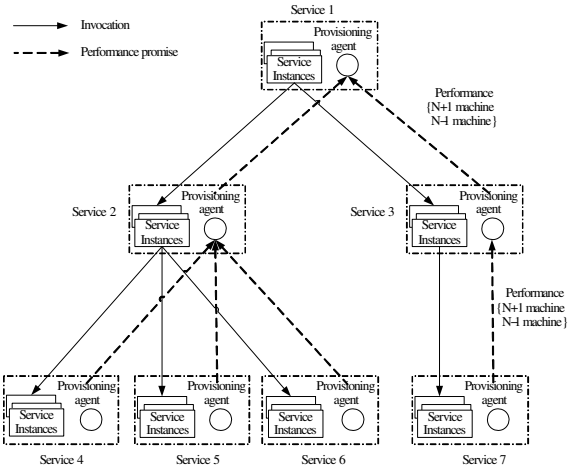


Figure 2: Resource provisioning system model

3. AUTONOMOUS PROVISIONING

3.1 System model

We define a service as either a single-tier functional service with an HTTP or SOAP interface hosted in an application server, or a single-tier data service with an SQL interface hosted in a database server. Although in real systems services may be composed of an application server and a database server, for provisioning we consider these as two separate services. Within a multi-service application, services are commonly organized as a directed acyclic graph. We assume that inter-service invocations are synchronous and that the services of one application are not used simultaneously by other applications (which means that the directed acyclic graph has a single root node).

Figure 1 shows how a service is typically hosted. A service may have multiple instances representing multiple application servers with a copy of the service code or multiple database servers containing a replica of the service's data. To improve performance, a service may possibly employ one or more machines as caches that intercept incoming requests before accessing the service itself. We use consistent hashing to distribute cached objects across multiple caches [8]. This means in particular that increasing the number of caches attached to a service generates the same hit rate as increasing the storage space of a single cache.

We assume that some machines are always available to be added to an application, as is commonly the case in clouds. Our system relies on an exclusive provisioning model: each resource can be assigned to only one service at a time. Such

resource may be a physical machine or a virtualized instance with performance isolation as for example in Amazon EC2.

Figure 2 illustrates our approach based on an invocation tree consisting of 7 services. Resource provisioning is done in two steps. First, each service carries out "what-if analysis" to predict its future performance in case it was assigned an extra machine or removed one. These prediction results can be seen as a performance promise made by the service to its parent in the invocation tree. Predictions are realized by a provisioning agent attached to each service. Each service periodically sends its performance promises to its parent in the invocation tree.

In the second step services negotiate resources with each other. Each intermediate node in the invocation tree negotiates performance with its parent on behalf of itself and all its children nodes. This intermediate service is responsible for all local resource provisioning decisions among its own subtree. A local decision consists of selecting the maximum performance gain (or minimum loss) among the service's children nodes and itself. For example, in Figure 2, services 2 and 3 report their performance promises to service 1 but the promise of service 2 is an aggregate among its own promises and those of services 4, 5 and 6.

Finally, the root node selects which service(s) to provision across the tree when the SLA is (about to be) violated, or to deprovision when this is possible without violating the SLA.

3.2 Performance model of a single service

A good performance model in our system should not only explain the current performance of a given concerned service, but also predict its future performance if one more or one less machine was assigned to host the service. Additionally, it should predict future performance in case its received request rate would increase or decrease. We first present the model itself, then discuss its parameterization.

3.2.1 Performance model

We model a single-core machine as an M/M/1/PS queue, which is widely adopted in practice [6]. Similarly, multi-core machines distribute their load evenly on each CPU core. Consequently, we use an M/M/n/PS queue to capture the performance of an n-core machine. We assume that all CPU cores of the provisioning machines are homogenous.

The performance model calculates the expected response time after adding or removing one server (such as application server or database server) as follows:

$$\Delta R_{server} = R(n \pm 1)_{server} - R(n)_{server}$$

$$R(n)_{server} = \frac{S_{server}}{1 - \frac{\lambda S_{server}}{n}}$$

where R_{server} is the average response time of the service, n is the number of CPU cores assigned to the service, λ is the average request rate and S_{server} is the mean service time of the server.

A service may also use caches to offload some of the incoming requests from the service itself. This is especially common in web applications when the request locality is high. On the other hand caches may also waste useful resources if the request locality is low. Adding caches potentially improves response time for two reasons. First, cache hits are processed faster than cache misses. Second, the service itself and all children nodes receive less requests, and

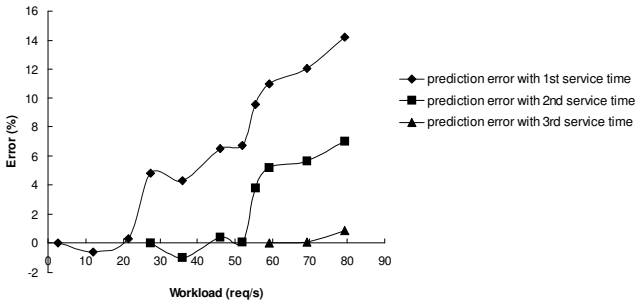


Figure 3: Dynamic service time correction

can thus process them faster. After adding a cache, the service response time consists of the cache fetching time and the sojourn time in the service upon every cache miss. The performance model calculates the caching impact on the response time as follows:

$$\Delta R_{cache} = R(n \pm 1)_{cache} - R(n)_{cache}$$

$$R(n)_{cache} = p_n S_{cache}(n) + (1 - p_n) R(m)$$

where $R(m)$ is the response time of the backend server across m CPU cores, S_{cache} is the cache service time, which is identical to the cache response time based on Little's Law [13], and p_n is the expected cache hit ratio with n nodes.

3.2.2 Model parameterization

Most of the model parameters can be measured offline or monitored at runtime. For example, the request rate can be monitored by the administrative tools of application servers and database servers. The cache service time can be obtained by measuring cache response time offline. However, the expected cache hit ratio p_n and the mean service time S_{server} are harder to measure.

We estimate the new cache hit ratio after a reconfiguration if one machine was added to or removed from the caching tier using virtual caches [12]. A virtual cache is a cache that stores only metadata such as the list of objects in cache and their sizes, but not the objects themselves. It receives all requests directed to the service and applies the same operations as a real cache with the same configuration would. It can thus estimate the hit ratio that a cache of any given size would have under the current workload.

Another crucial parameter is the service time S_{server}. Previous research works measure the service time via profiling under low workload [12, 15]. However, we found that the service time changes under different workloads, probably because of extra overhead in the server implementation that is not captured by an M/M/n/PS queue. We illustrate this in Figure 3. We first measure the service time of a database service under a low workload of 1 req/s. We then use this value to predict the response time of the service under other workloads. The curve with the diamond label indicates the prediction error under various workloads compared to the corresponding measured value. The error initially remains close to 0. However it later increases and finally reaches 14%, which is not acceptable for our purpose.

To achieve acceptable prediction results, we apply a classical feedback control loop to adjust the service time at runtime. The system continuously estimates the service's response time under the current conditions and compares the

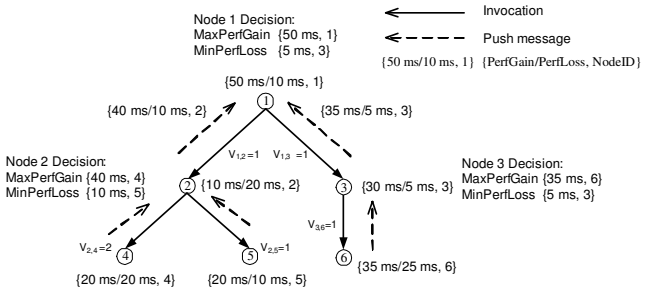


Figure 4: Provisioning in hierarchical structures

error between the predicted response time and the measured one. One can define a threshold as a configuration parameter. When the prediction error exceeds the threshold, the correction mechanism recomputes the service time:

$$S'_{server} = \frac{nR_{server}}{n + \lambda R_{server}}$$

where S'_{server} is the corrected service time, R_{server} is the latest measured response time, n is the number of current CPU cores, and λ is the current request rate.

Figure 3 shows the effectiveness of this mechanism. We define the error threshold as 5% and apply the correction mechanism to the whole prediction process carried out for the curve with diamond label. When the workload reaches 28 req/s, the prediction error exceeds the threshold. The control system then recomputes the service time. The curve with the square label presents the prediction error with this second service time value. Compared with the error caused by the original service time measured offline, this corrected service time causes much fewer error when the workload increases further. Similarly, the control system triggers the correction again around 53 req/s. The curve with the triangle label displays the further prediction error, which is again within the limits.

The system also maintains a memory of the service time values that should be used for various workload intensities.

3.3 Resource provisioning of service instances

Resource provisioning within a multi-service application is based on negotiation among services, where services continuously exchange performance promises generated by the performance model. We first discuss the case where services are organized in a tree pattern and only service instances are added or removed, then extend to directed acyclic graphs.

3.3.1 Hierarchical structure

Multi-service applications are often organized along a hierarchical structure. To find out which service(s) should be reprovisioned, services exchange their future performance objectives if a resource reconfiguration would happen. Each service reports performance promises to its parent on behalf of its children and itself: it reports the best performance gain (resp. loss) possible by adding (resp. removing) a server to (resp. from) a service of the subtree consisting of its children nodes and itself.

Figure 4 illustrates the decision processes within a typical hierarchical structure. The decision process between service 2 and its children 4 and 5 is the smallest decision unit in the whole application. Here, services 4 and 5 are responsible for reporting their performance promises to service 2. To generate its own promises, service 2 must find the maximum performance gain (resp. minimum loss) that the entire subtree can achieve with one more (resp. one less) machine. Assuming a service i has k immediate children services, it aggregates its own performance promises as follows:

$$MaxPerfGain = max\{V_{i,j} \cdot MaxPerfGain_j\} \quad (1 \le j \le k)$$

$$MinPerfLoss = min\{V_{i,j} \cdot MinPerfLoss_j\} \quad (1 \le j \le k)$$

where $V_{i,j}$ is the average number of service executions on service j caused by one request from service i. For example, in Figure 4 each request from service 2 results on average in two service executions on service 4 and only one on services 5. This parameter can be measured online by services 4 and 5 by comparing their local request rate with that of their parent. The children nodes adaptively adjust this parameter when they observe that the ratio changes. Here, although service 4 would gain 20 ms if it was given one more machine, the actual performance gain brought by service 4 to service 2 is 40 ms due to the double invocation ratio. Service 2 compares the received promises with its own local ones, and makes the local decision: if given one more machine, it should give it to service 4 which generates the greatest performance gain overall. If requested to release one machine, it should remove it from service 5 which would incur the lowest global performance loss.

The same process is repeated at every level of the tree up to the root, which has sufficient information to take provisioning decisions upon variations in request rate. Here, service 1 can finally make the global decision: if given one more machine, it should keep it to itself as it can obtain the maximum performance gain. If removing one machine, it should remove it from service 3 as this causes minimum performance loss. Once service 1 decides to change resource allocation, it triggers the reconfiguration by sending a notification to the concerned service.

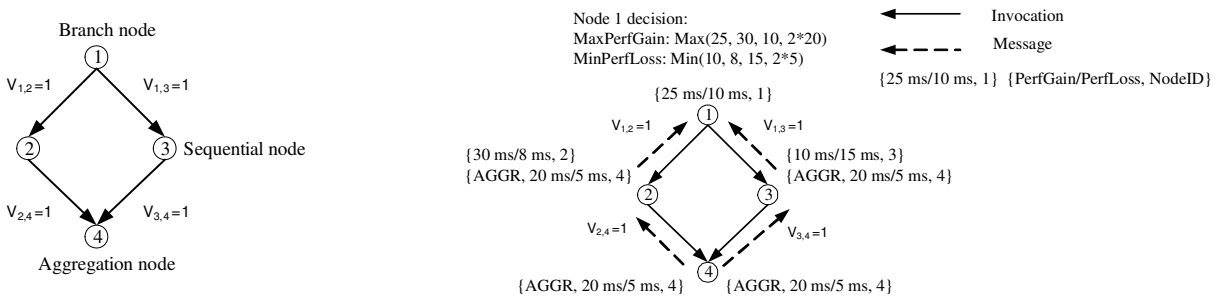
3.3.2 Directed acyclic invocation graphs

In real-world applications, multiple services may commonly share the same backend. For example, in Figure 5(a) service 4 may be a database accessed by multiple web services. We define a shared service as an aggregation node in the invocation path.

Any performance promise made by an aggregation node or any of its children has the same effect to each invocation path from the root node to the aggregation node. For example, in Figure 5(a), there are two invocation paths from root node 1 to aggregation node 4: $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$. Assuming that aggregation node 4 would gain 20 ms due to its own resource reconfiguration, then root node 1 would gain a total performance improvement of 40 ms. The negotiation mechanism should reflect the multi-path performance effect of the aggregation node.

Aggregation nodes report their promises along each invocation path with special "AGGR" identifications. In Figure 5(b), service 4 sends {AGGR, 20 ms/5 ms, 4}. This means service 4 is an aggregation node, and would gain 20 ms with an extra machine or lose 5 ms with one less machine.

"AGGR" messages are handled differently than regular promises. Any node receiving such message should forward



(a) Directed acyclic invocation graph

(b) Resource provisioning process

Figure 5: Resource provisioning in directed acyclic graphs

it upwards to the root in addition to the regular promises. If a node receives multiple "AGGR" messages originating from the same node ID, it must add them together before forwarding. Finally the root node compares performance promises from regular messages and the ones from "AGGR" messages to make its global decision. For example, in Figure 5(b), service 1 receives two "AGGR" messages with the same node ID 4. It thus adds them as the performance promise of service 4. As service 1 is the root node, it also compares other performance promises with the merged result 2*20 ms, and finds the maximum one as the final decision.

3.4 Resource provisioning of cache instances

Thus far we only discussed provisioning of service instances. Provisioning cache instances is harder because it not only changes the performance of the concerned service, but also changes the traffic to its children, which in turn affects their performance. Thus, each service should also calculate the performance it would have if addressed more or less traffic.

When considering whether to add or remove a cache to itself (instead of a service instance), each service must take into account the future expected performance of all its children services if they would receive more/less traffic.

In our system, each node operates two virtual caches with different sizes matching the situations where the service would be assigned one more or one less cache instance. Each service periodically informs its children of the relative workload decrease (resp. increase) it would address to them if it was given one more (resp. one less) cache instance. This expected invocation ratio EIR on the node originating cache reconfiguration is equal to the expected miss rate:

$$EIR = ExpectedMissRate$$

In such case the children can anticipate a decrease or increase of the traffic they receive. We illustrate this information exchange process for cache effect calculation in Figure 6(b), which features a complex situation with multiple aggregation nodes. When a node j receives the "CACHE"-labeled messages including expected invocation ratios from its parents, it first computes its local expected workload intensity as the sum of expected request rates promised by its predecessors:

$$w_j^{'} = \sum_{i=1}^{k} V_{i,j} * EIR_i * w_i$$

where $V_{i,j}$ is the average number of service executions on service j caused by one request from service i, EIR_i is the expected invocation ratio of parent i, w_i is the request rate of node i, and k is the number of its predecessors in the invocation graph. Then the node calculates its own expected invocation ratio:

$$EIR_j = \frac{w_j^{'}}{w_j}$$

For example, in Figure 6(b), node 4's expected invocation ratio is:

$$EIR_4 = \frac{w_4^{'}}{w_4} = \frac{w_2 + EIR_3 * w_3}{w_4}$$

The concerned node j forwards its expected invocation ratio EIR_j to its children, then calculates its own expected performance under its expected workload intensity. Finally, it returns calculated performance objectives to all its parents.

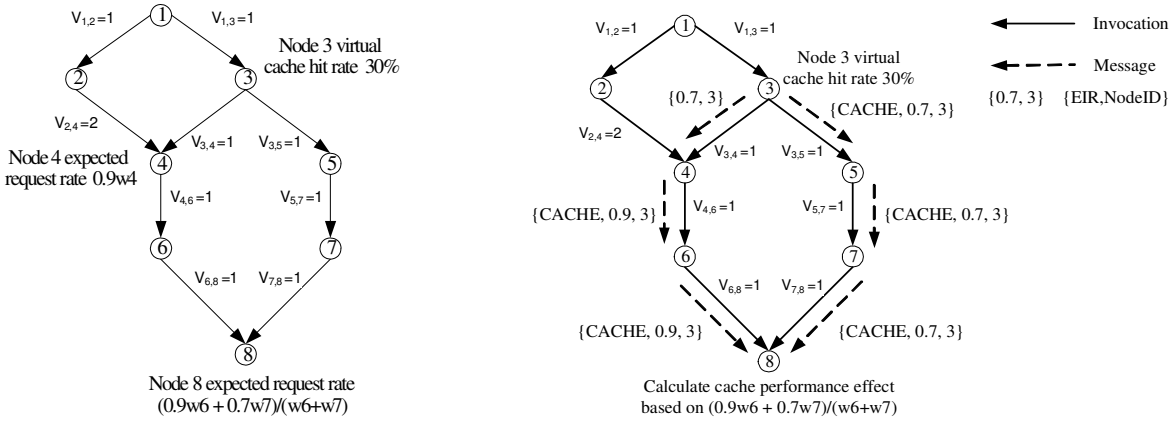
In a directed acyclic graph, a performance change in an aggregation node affects all its predecessor branches but also other branches as well. For example, adding a cache to service 2 in Figure 6(b) would change the performance of service 4, and thereby also affect service 3. The "AGGR" messages are also employed to propagate information about these cascading effect through the invocation graph.

Note that, even though the system may need to propagate many "AGGR" messages simultaneously, there is no combinational explosion: in the worst case, the number of "AGGR" messages processed by a node is linear to the number of nodes in the invocation graph.

3.5 Shifting resources among services

In many cases, instead of provisioning extra resources, it can be more efficient to simply reorganize resource assignments within the application without retrieving machines from the resource pool. Such reorganization may be necessary to follow changes in access patterns. For example, in Figure 4, the values $V_{2,4}$, $V_{2,5}$ and $V_{2,6}$ may change due to an update in the application code or a change in user behavior. Our system should reorganize the resource assignments so as to increase resource usage, and therefore improve application performance.

One could imagine letting each intermediate service shift resources autonomously within its children and itself. However, this could lead to inefficiencies such as having the application from Figure 4 shift resources from service 4 to



(a) Complex directed acyclic graph

(b) Cache effect calculation

Figure 6: Cache instance provisioning in complex directed acyclic graphs

service 5 (initiated by service 2), immediately followed by shifting the same resource again from service 5 to service 3 (initiated by service 1). We therefore prefer letting only the root node be responsible for such reconfigurations.

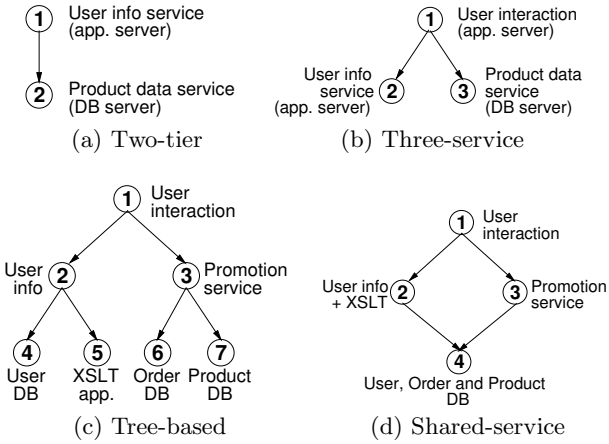
To prevent oscillating behavior, one should first define a performance improvement threshold as the criterion for deciding whether to shift resources. In a hierarchical invocation case, each service should compose its performance objectives in case one machine was shifted from the service having minimum performance loss to the one having maximum performance gain within the tree. These promises can get aggregated up in the invocation graph such that the root node finally selects the greatest reorganization performance promise and triggers the reconfiguration

In a directed acyclic graph, only the root node has complete information about performance promises from "AGGR" messages. Any node receiving "AGGR" messages does not compose performance objective for shifting resources. Instead, the root node is finally responsible for finding the maximum performance gain and minimum performance loss and composing these two values as the performance objective of shifting resources from the global perspective.

Note that when one shifts a cache resources upwards within the same invocation path, the affected node to which resources are shifted changes the traffic to all its children and itself. To help generate performance promises in this special case, each service should send expected performance objectives if addressed with the expected request rate. In the case a service shifted one machine upwards to any service in the same path as a cache, this service would serve requests with one less machine. Therefore, each service should send its expected performance objectives under the expected request rates on both the original resource configuration and the updated one.

4. EVALUATION

This section first validates the performance model discussed in Section 3.2. We then compare our system with two representatives of the state of the art. Finally, we demonstrate the unique features of our approach for provisioning directed acyclic graphs of services querying each other.



(a) Two-tier

(b) Three-service

(c) Tree-based

(d) Shared-service

Figure 7: Web applications under test

4.1 Experimental setup

We evaluate our system using four reference applications depicted in Figure 7. Figure 7(a) shows a classical two-tier application. The application server tier receives HTTP requests and issues one query to the database to search for items related to the last ones purchased by the concerned client. It then applies CPU-intensive XSLT transformation to transform XML templates into HTML.

Figure 7(b) shows a three-service application with similar features to the first application. Here, however, the "User interaction" servlet first invokes the "User info" service through a SOAP interface and then the "Promotion" data service through a SQL interface.

The application in Figure 7(c) follows a strict tree-like invocation pattern. The root service invokes the left branch for gathering user information, then the right branch for promoting product information to the same user. The "User info" service in turn accesses user data from the "User" data service, then invokes an external "XSLT" service to transform XML templates into HTML. The "Promotion" service in the right branch first fetches users' order histories from the "Order" data service, then searches for items related to users' last orders using the "Product" data service in order

Table 1: Model validation for XSLT service

# App. servers	# Caches	Request rate	Measured resp. time	Predicted resp. time
1	0	36 req/s	488.3 ms	N/A
1	1	36 req/s	172.3 ms	177.1 ms (+2.8%)
2	0	36 req/s	111.0 ms	**116.0 ms (+4.5%)**
2	1	90 req/s	125.6 ms	**131.2 ms (+4.4%)**
3	0	90 req/s	135.1 ms	139.8 ms (+3.5%)

Table 2: Model validation for Product service

# DB servers	# Caches	Request rate	Measured resp. time	Predicted resp. time
1	0	10 req/s	449.0 ms	N/A
1	1	10 req/s	209.0 ms	**219.0 ms (+4.8%)**
2	0	10 req/s	263.1 ms	271.4 ms (+3.2%)
1	2	18 req/s	111.6 ms	**112.7 ms (+1.0%)**
2	1	18 req/s	199.8 ms	201.8 ms (+1.0%)

to recommend further purchases. Finally, the root service combines the results from the two branches in one web page and returns it to the client.

The last application in Figure 7(d) is similar to the third one but is structured so that all "User", "Order" and "Product" data are stored in a single, shared data service. The "User info" service also handles the XML transformation.

In all experiments, we emulate various numbers of end-user browsers which send requests to the applications with Poisson distribution of arrival times. This distribution has been shown to be realistic for many Internet systems [17]. We implement the local performance monitor on application server using the MBean servlet from JBoss. The database server monitoring is based on performance data collected by the admin tool of MySQL. We developed the negotiation agent in Java using plain sockets.

All experiments are performed on the DAS3 cluster at VU University Amsterdam [3]. This cluster consists of 85 nodes, each of which having a dual-CPU/dual-core 2.4GHz AMD Operon DP 280, 4GB RAM and a 250 GB IDE hard drive. Nodes are connected with a 10Gbps LAN such that the network latency between nodes is negligible. During the whole experiments, we set the prediction error threshold for dynamically adjusting the service time to 3%.

4.2 Model validation for single service

Before focusing on resource provisioning, we first validate our performance model using the "XSLT" and "Product" services from Figure 7(c) separately. The two services are respectively application server-intensive and database server-intensive. We set the SLA of each service to a maximum response time of 400 ms, and initially assign one server to each. We then increase the request rates until the SLA is violated. At that time, we issue performance predictions in case one more machine was assigned as a server replica or a cache, and compare predicted values with the measured response times after applying adaptations. Tables 1 and 2 show the results at two prediction points for the two services separately.

The first SLA violation of the "XSLT" service occurs around 36 req/s. Prediction errors of adding a server replica and adding a cache are under 5%. Results clearly show that adding a second server is more efficient in this case. We perform the adaptation and increase workload until 90 req/s when the SLA is violated again. Here as well the prediction errors remain under 5%. Similarly, the first SLA violation of the "Product" service occurs around 10 req/s. Both prediction errors are also under 5%. We add a cache to the service and increase workload until 18 req/s when the SLA is violated again. The prediction errors again remain very low, which confirms the accuracy of our model.

4.3 Comparison with the state of the art

We now compare our system with two representatives of the state of the art in resource provisioning. One of the most cited papers on resource provisioning for multi-tier applications is [14]. This approach is based on analytic models and thus we name it "Analytic" in this section while we name our system "Autonomous" here. The second approach assigns a fixed SLA to each service individually.

4.3.1 Comparison with Analytic

Analytic is designed to provision resources in multi-tier web applications such as the two-tier application from Figure 7(a). We here demonstrate that both schemas work equally well for such applications. We assign an SLA of 500 ms for the whole application, and initially assign one application server and one database server to the application. As the performance model in [14] is based on single-core single-CPU machines, we run our experiments using only one core of each hosting machine on the DAS3 cluster.

We provision both servers and caches to the tested web application. We increase the workload to obtain two successive adaptations. We record provisioning decisions of each schema and compare their predicted response times with the measured ones. Table 3 shows that both schemas issue slightly different performance predictions but take the same provisioning decisions: at 12 req/s, both systems add an application server to the application. At 18 req/s, both systems add a database server.

In all cases the prediction errors are lower than 7%, which confirms that both approaches can provision multi-tier applications with similar accuracy. On the other hand, Analytic does not address multi-service applications organized in hierarchical or directed acyclic graph patterns.

4.3.2 Comparison with per-service SLA

We now compare our system with the per-service SLA approach. This approach is popular in complex multi-service applications as in Figures 7(c) and 7(d). However, we claim that it often uses unnecessary resources due to the impossibility of defining suitable SLAs for internal services. We illustrate this using the application in Figure 7(b). We define the global application SLA as 500 ms, and run the two systems across three successive adaptations. For simplicity, in this section we do not consider cache provisioning.

We first use our system to provision the test application. As shown in Figure 8(a), our system adds an application server to service 2 at 12 req/s, then a database server to service 3 at 16 req/s, and finally another application server to service 2 at 25 req/s.

We now show that it is impossible to give a fixed SLA to service 2 such that the per-service SLA approach takes optimal provisioning decisions. We set the SLA of the front-end service to 500 ms, identical to the SLA of the whole application. The best possible SLA for service 2 in this case is 290 ms: it allows the system to reprovision service 2 at

Table 3: Resource provisioning of two-tier web application

# APP servers	# DB servers	# App caches	# DB caches	Request rate	Measured resp. time	Autonomous prediction	Analytic prediction
1	1	0	0	12 req/s	552.6 ms	N/A	N/A
2	1	0	0	12 req/s	303.5 ms	**309.1 ms (+1.8%)**	**296.2 ms (-2.4%)**
1	2	0	0	12 req/s	419.8 ms	443.0 ms (+5.5%)	447.2 ms (+6.5%)
1	1	1	0	12 req/s	515.3 ms	543.2 ms (+5.4%)	503.4 ms (-2.3%)
1	1	0	1	12 req/s	405.0 ms	391.4 ms (-3.4%)	389.8 ms (-3.8%)
2	1	0	0	18 req/s	511.2 ms	N/A	N/A
3	1	0	0	18 req/s	481.4 ms	473.5 ms (-1.6%)	491.1 ms (+2.0%)
2	2	0	0	18 req/s	210.1 ms	**223.3 ms (+6.3%)**	**197.3 ms (-6.1%)**
2	1	1	0	18 req/s	498.7 ms	505.2 ms (+1.3%)	507.2 ms (+1.7%)
2	1	0	1	18 req/s	241.4 ms	230.6 ms (-4.5%)	243.2 ms (+0.7%)

Table 4: Prediction accuracy under increasing workload for tree application

	Add a cache at 10 req/s	Add a server at 10 req/s	Add a cache at 18 req/s	Add a server at 18 req/s
Serv. 1	520.9 ms (-1.1%)	522.5 ms (-1.3%)	492.2 ms (+1.2%)	500.8 ms (+1.6%)
Serv. 2	518.6 ms (+0.6%)	524.1 ms (-0.4%)	481.0 ms (+2.0%)	496.3 ms (+2.6%)
Serv. 3	493.3 ms (+2.3%)	501.9 ms (+1.9%)	503.3 ms (+1.8%)	510.8 ms (+1.9%)
Serv. 4	525.5 ms (-1.0%)	525.0 ms (-1.0%)	511.4 ms (+1.4%)	507.9 ms (+1.6%)
Serv. 5	489.2 ms (+2.9%)	409.1 ms (+3.3%)	453.2 ms (+3.0%)	**401.9 ms (+2.8%)**
Serv. 6	525.1 ms (-1.0%)	524.8 ms (-1.2%)	518.6 ms (+1.0%)	508.0 ms (+1.1%)
Serv. 7	**305.3 ms (+5.0%)**	399.1 ms (+3.8%)	449.5 ms (+2.0%)	463.8 ms (+1.6%)

12 req/s (which we know to be the optimal decision in this case), just before the application would violate its global SLA. Similarly, we set the SLA of service 3 to 365 ms.

Figure 8(b) shows the performance of the per-service SLA approach. The first two adaptations are identical to those of our own system. However, at 23 req/s service 2 violates its internal SLA although the application as a whole does not violate the global SLA. The per-service SLA strategy therefore adds a server to service 2 at 23 req/s, which is wasteful between 23 req/s and 25 req/s.

Selecting other values for the internal SLAs leads to even worse performance. If the SLA of service 2 was set lower than 290 ms, then the per-service SLA approach would reprovision service 2 too early at the first adaptation already. On the other hand, if its internal SLA was set to a greater value than 290 ms, then at the first adaptation this strategy would reprovision the front-end instead of service 2, which does not gain enough performance to maintain the application within its global SLA.

The per-service SLA approach allows one to provision arbitrary multi-service applications. However, even when configured with the best possible internal SLA values, it uses more resources than our proposed system.

4.4 Provisioning of multi-service applications

We now illustrate the unique features of our system using the tree-based application from Figure 7(c) and the shared-service one from Figure 7(d). We set the SLA to 500 ms.

4.4.1 Provisioning under varying load intensity

Figure 9 shows the response time of the two applications when their request rates vary. Figure 9(a) depicts the test scenario: the workload first increases from 2 req/s to 22 req/s, then decreases back to 2 req/s.

Figure 9(b) shows the performance of the tree-based application. Our system adds resources twice at 10 req/s and 18 req/s, adding a cache to service 7 then an application server to service 5. When the workload decreases, opposite decisions are taken at 16 req/s and 8 req/s. Figure 9(c) shows similar results for the shared-service application.

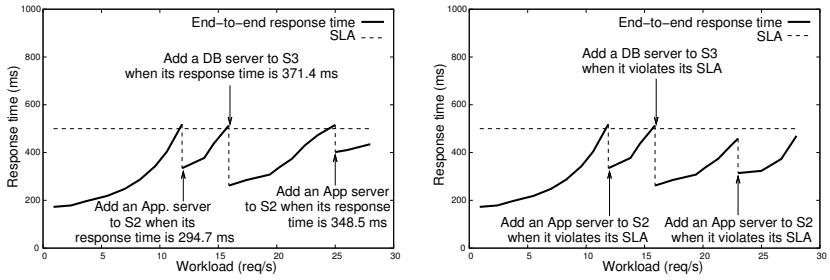
For all reconfigurations proposed by the provisioning system, we also verify the decisions by measuring the end-to-end response time of all other possible adaptations. Tables 4 and 5 show the prediction accuracy under increasing workload for the two applications at their respective adaptation points. In all cases the predictions remain very accurate and allow one to make the optimal provisioning decision. Similar accuracy is also obtained when decreasing the workload.

These results show that our provisioning system can correctly identify the most bottlenecked service within entire tree-based or shared-service applications when their SLA targets are violated. Meanwhile, our system can also save resource usage by removing resources from the least affected service while remaining within the SLA.

4.4.2 Provisioning under varying load distribution

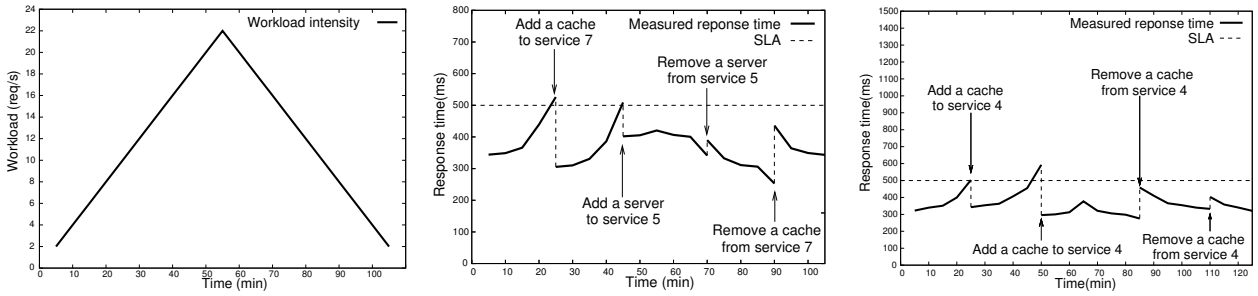
We now turn to more subtle cases where the front-end's request rate remains stable but internal parameters such as the invocation count from one service to another changes over time. The relative utilization of assigned resources may thus change over time. Figure 10(a) depicts the scenario for the tree-based application: the workloads of services 2 and 3 first increase at the same rate. At time 35, the workload of service 3 drops by a factor 10, while service 2 maintains the same increase rate. We apply a similar scenario to services 2 and 3 of the shared-service application. In these experiments, we set the performance improvement threshold before shifting resources to 30%.

Figure 10(b) shows the behavior of the tree-based application. At time 25, our system proposes to add one cache to service 7 due to an SLA violation. At time 40, the response time of the whole application drops because less traffic is issued to service 3. Then the response time increases again. At time 70, the SLA is not violated but our system decides to shift one machine from service 7 to service 5 so as to gain 30% performance improvement with better resource



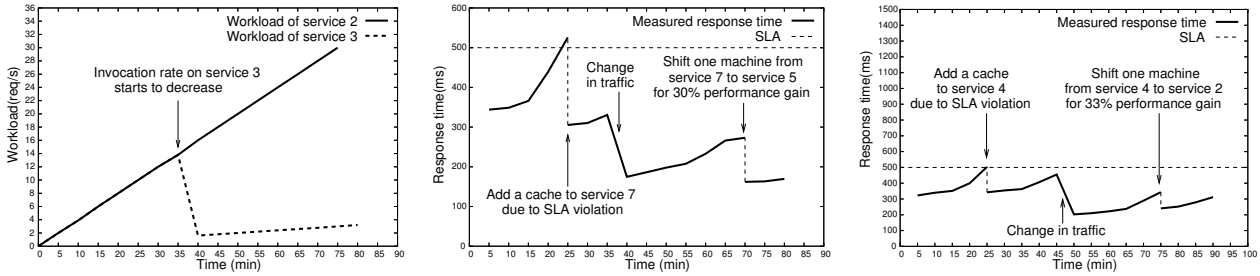
(a) Autonomous provisioning (b) Per-service SLA provisioning

Figure 8: Comparison between our system and per-service SLA



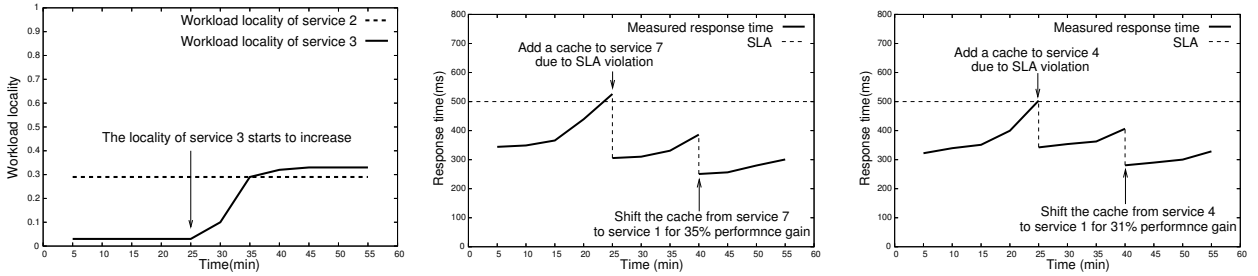
(a) Experiment scenario (b) Tree-based application (c) Shared-service application

Figure 9: Resource provisioning under varying load intensity



(a) Experiment scenario (b) Tree-based application (c) Shared-service application

Figure 10: Resource provisioning under varying load distribution



(a) Experiment scenario (b) Tree-based application (c) Shared-service application

Figure 11: Resource provisioning under varying load locality

Table 5: Prediction accuracy under increasing workload for shared-service application

	Add a cache at 7.5 req/s	Add a server at 7.5 req/s	Add a cache at 15 req/s	Add a server at 15 req/s
Serv. 1	400.2 ms (+1.5%)	493.6 ms (+2.1%)	428.1 ms (+2.2%)	481.3 ms (+0.9%)
Serv. 2	447.7 ms (+3.7%)	469.7 ms (+1.1%)	378.6 ms (-1.3%)	409.1 ms (+2.3%)
Serv. 3	465.3 ms (+1.3%)	489.1 ms (+1.8%)	452.7 ms (+0.8%)	473.4 ms (-0.7%)
Serv. 4	**342.3 ms (+3.9%)**	375.2 ms (+4.8%)	**295.7 ms (+2.7%)**	413.9 ms (+2.1%)

organization. Figure 10(c) shows similar behavior for the shared-service application.

These results show that we can optimize resource organization without retrieving extra resources by identifying potential improvements of resource utilization.

4.4.3 Provisioning with varying load locality

Another subtle form of change in workload is a variation of workload locality. Here, the potential performance of a cache varies over time. We define the locality as the hit rate for a cache holding 10,000 objects. Figure 11(a) depicts the evaluated scenario for the tree-based application: we first increase the workload until time 25 when the end-to-end response time violates the SLA target. Immediately after reconfiguration, we start changing the locality of service 3. We apply a similar scenario to the shared-service application.

Figure 11(b) shows that the tree-based application first adds one cache to service 7 at time 25. When the locality of service 3 changes, our system shifts the cache from service 7 to become a cache in service 1, such that the end-to-end response time improves by 35%. Figure 11(c) shows similar results for the shared-service application.

These results show that we can reorganize the cache assignment within a whole application to adapt to changes in traffic locality and improve application performance.

5. CONCLUSIONS

Most Web resource provisioning approaches rely on a single analytical queuing model to capture the application's performance features. However, applying such approaches to multi-service web applications is a challenge due to complex service relationships and the cascading effects of caching. This paper takes a different stand and demonstrates that provisioning resources for multi-service applications can be achieved in a decentralized way where each service is autonomously responsible for its own provisioning.

We propose to give an SLA only to the front-end service. All other services collaboratively negotiate their future performance objectives with each other to make provisioning decisions. Resource provisioning is based on "what-if analysis" where each service continuously reports its performance promises in case if it was assigned more/less resources, or if it received more/less traffic. The negotiation process occurs recursively between levels of the whole invocation graph. The root node is responsible for selecting service(s) to provision so as to maintain the front-end service's SLA and maximize resource utilization.

We demonstrated through extensive experiments that our scheme allows one to capture the cascading effects of resource provisioning in multi-service applications. To our best knowledge, no other published resource provisioning algorithm can match or outperform our approach. Should major Web hosting businesses adopt our techniques instead of a fixed SLA per service, they could drive accurate resource provisioning at lower costs.

6. REFERENCES

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: a control-theoretical application. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), 2002.

[2] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian. Resource management in the autonomic service-oriented architecture. In *Proc. ICAC*, 2006.

[3] DAS3: The Distributed ASCI Supercomputer 3. `http://www.cs.vu.nl/das3/`.

[4] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a Web service utility. In *Proc. USITS*, 2003.

[5] J. Gray and W. Vogels. A conversation with Werner Vogels. *ACM Queue*, 4(4), 2006.

[6] N. J. Gunther. *Analyzing Computer System Performance with Perl::PDQ*. Springer, 2005.

[7] A. Kamra, V. Misra, and E. M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites. In *Proc. Intl. Workshop on Quality of Service*, 2004.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. ACM Symposium on Theory of Computing*, 1997.

[9] H. A. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proc. EuroSys*, 2009.

[10] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-driven server migration for Internet data centers. In *Proc. Intl. Workshop on Quality of Service*, 2002.

[11] R. Shoup. eBay's architectural principles. `http://jaoo.dk/london-2008/file?path=/qcon-london-2008/slides/RandyShoup_eBaysArchitecturalPrinciples.pdf`.

[12] S. Sivasubramanian. *Scalable hosting of web applications*. PhD thesis, VU University Amsterdam, Netherlands, 2007.

[13] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, 2001.

[14] 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.

[15] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous Adaptive Systems*, 3(1), 2008.

[16] T. Vercauteren, P. Aggarwal, X. Wang, and T.-H. Li. Hierarchical forecasting of Web server workload using sequential Monte Carlo training. In *Proc. Conf. on Information Sciences and Systems*, 2006.

[17] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transactions on Internet Technology*, 7(1), 2007.

[18] B. Y. Wu, C. H. Chi, and Z. Chen. Resource allocation based on workflow for enhancing the performance of composite service. In *Proc. SCC*, 2007.