Vertical Load Distribution for Cloud Computing via Multiple Implemention Options

Thomas Phan and Wen-Syan Li

Abstract Cloud computing looks to deliver software as a provisioned service to end users, but the underlying infrastructure must be sufficiently scalable and robust. In our work, we focus on large-scale enterprise cloud systems and examine how enterprises may use a service-oriented architecture (SOA) to provide a streamlined interface to their business processes. To scale up the business processes, each SOA tier usually deploys multiple servers for load distribution and fault tolerance, a scenario which we term horizontal load distribution. One limitation of this approach is that load cannot be distributed further when all servers in the same tier are loaded. In complex multi-tiered SOA systems, a single business process may actually be implemented by multiple different computation pathways among the tiers, each with different components, in order to provide resilience and scalability. Such multiple implementation options gives opportunities for vertical load distribution across tiers. In this chapter, we look at a novel request routing framework for SOA-based enterprise computing with multiple implementation options that takes into account the options of both horizontal and vertical load distribution.

1 Introduction

Cloud computing looks to have computation and data storage moved away from the end user and onto servers located in data centers, thereby relieving users of the burdens of application provisioning and management [7, 2]. Software can then be thought of as purely a service that is delivered and consumed over the Internet, offering users the flexibility to choose applications on-demand and allowing providers to scale out their capacity accordingly.

Thomas Phan
Microsoft Corporation, USA, e-mail: thomas.phan@acm.org

Wen-Syan Li
SAP Technology Lab, China, e-mail: wen-syan.li@sap.com

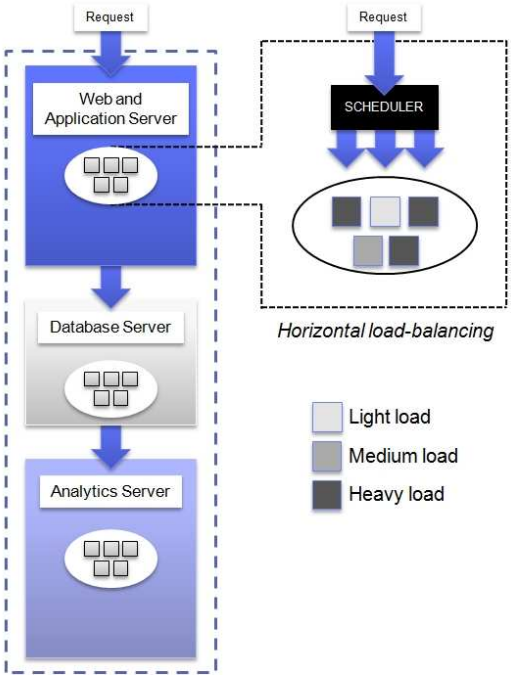

Fig. 1 Horizontal load distribution: load is distributed across a server pool within the same tier.

As rosy as this picture seems, the underlying server-side infrastructure must be sufficiently robust, feature-rich, and scalable to facilitate cloud computing. In this chapter we focus on large-scale enterprise cloud systems and examine how issues of scalable provisioning can be met using a novel load distribution system.

In enterprise cloud systems, a service-oriented architecture (SOA) can be used to provide a streamlined interface to the underlying business processes being offered through the cloud. Such an SOA may act as a programmatic front-end to a variety of building-block components distinguished as individual services and their supporting servers (e.g. [8]). Incoming requests to the service provided by this composite SOA must be routed to the correct components and their respective servers, and such routing must be scalable to support a large number of requests.

In order to scale up the business processes, each tier in the system usually deploys multiple servers for load distribution and fault tolerance. Such load distribution across multiple servers within the same tier can be viewed as *horizontal* load distribution, as shown in Figure 1. One limitation of horizontal load distribution is that load cannot be further distributed when all servers in the given tier are loaded as a result of mis-configured infrastructures – where too many servers are deployed at one tier while too few servers are deployed at another tier.

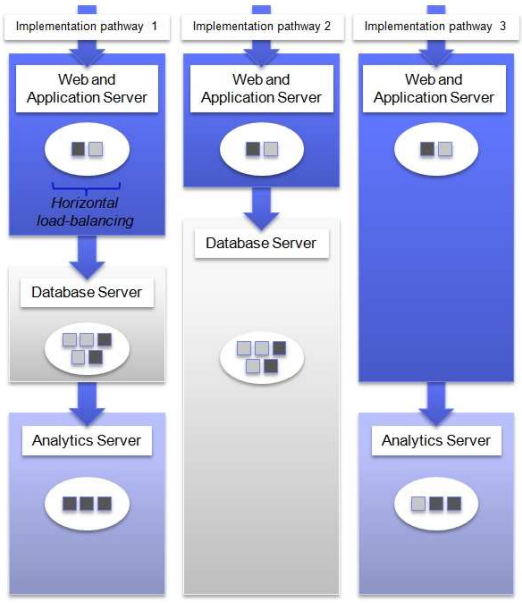

Fig. 2 Vertical load distribution: load can be spread across multiple implementations of the same composite service. This figure illustrates three different implementations of the same service that was shown in Figure 1.

An important observation is that in complex multi-tiered SOA systems, a single business process can actually be implemented by multiple different computation pathways through the tiers (where each pathway may have different components) in order to provide resiliency and scalability. Such SOA-based enterprise computing with multiple implementation options gives opportunities for *vertical* load distribution across tiers.

Although there exists a large body of research and industry work focused on request provisioning by balancing load across the servers of one service [5, 17], there has been little work on balancing load across *multiple implementations of a composite service*, where each service can be implemented via pathways through different service types.

A composite service can be represented as multiple tiers of component invocations in an SOA-based IT infrastructure. In such a system, we differentiate *horizontal* load distribution, where load can be spread across multiple servers for one service component, from *vertical* load distribution, where load can be spread across multiple implementations of a given service. The example in Figure 2 illustrates these terms. Here a composite online analytic task can be represented as a call to a Web and Application Server (WAS) to perform certain pre-processing, followed by a call from the WAS to a database server (DB) to fetch required data set, af-

ter which the WAS forwards the data set to a dedicated analytic server (AS) for computationally-expensive data mining tasks.

This composite task can have multiple implementations in a modern IT data center. An alternative implementation may invoke a stored procedure on the database to perform data mining instead of having the dedicated analytic server perform this task. This alternative implementation provides *vertical* load distribution by allowing the job scheduler to select the WAS-and-DB implementation when the analytic server is not available or heavily loaded. Multiple implementations are desirable for the purpose of fault tolerance and high flexibility for load balancing. Furthermore, it is also desirable for a server to be capable of carrying out multiple instances of the same task for the same reasons.

Reusability is one of the key goals of the SOA approach. Due to the high reusability of application components, it is possible to define a complex workflow in multiple ways. However, it is hard to judge in advance which one is the best implementation, since in reality the results depend on the runtime environment (e.g. what other service requests are being processed at the same time). We believe that having multiple implementations provides fault tolerance and scalability, in particular when dealing with diverse runtime conditions and missed configured infrastructures. In this respect, an SOA plays an important role in enabling the feasibility and applicability of multiple implementations.

In this chapter we propose a framework for request-routing and load balancing *horizontally and vertically* in SOA-based enterprise cloud computing infrastructures. We show that a stochastic search algorithm is appropriate to explore a very large solution space.

In our experiments, we show that our algorithm and methodology scale well up to a large scale system configuration comprising up to 1000 workflow requests to a complex composite web services with multiple implementations. We also show that our approach that considers both *horizontal and vertical* load distribution is effective in dealing with a misconfigured infrastructure (i.e. where there are too many servers in one tier and too few servers in another tier).

The key contributions of this paper are the following:

- We identify the need for QoS-aware scheduling in workloads that consist of composite web services. Our problem space lies in the relationship between consumers, service types, implementation options, and service providers.
- We provide a framework for handling both *horizontal and vertical* load distribution.
- We provide a reference implementation of a search algorithm that is able to produce optimal (or near-optimal) schedules based on a genetic search heuristic [12].

The rest of this chapter is organized as follows. In Section 2, we describe the system architecture and terminology used in this paper. In Section 3, we describe how we model the problem and our algorithms for scheduling load distribution for composite web services. In Section 4 we show experimental results, and in Section 5 we discuss related work. We conclude the paper in Section 6.

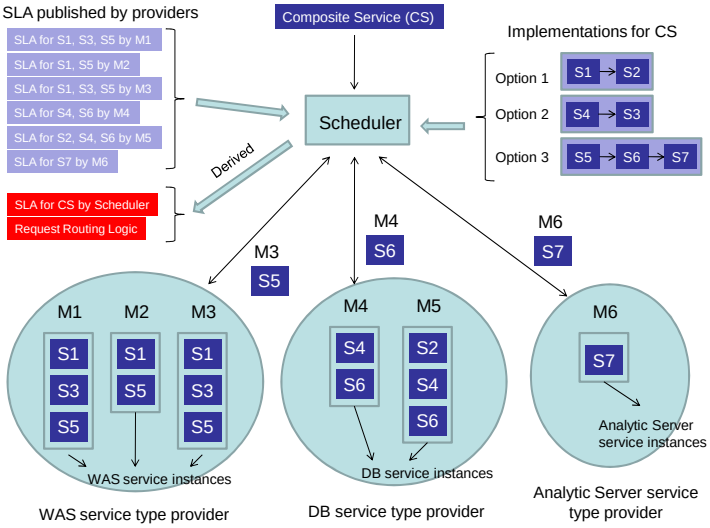

Fig. 3 Request routing for SOA-based enterprise computing with multiple implementation options.

2 Overview

In this section we give a system architecture overview and discuss the terms that will be used in this paper. Consider a simplified cloud computing example (shown in Figure 3) in which an analytic process runs on a Web and Application Server (WAS), a Database Server (DB), and a specialized Analytic Server (AS). The analytic process can be implemented by one of three options (as shown in the upper-right of the figure):

- Executing some lightweight pre-processing at WAS (S_1) and then having the DB to complete most of expensive analytic calculation (S_2); or
- Fetching data from the DB (S_4) to the WAS and then completing most of the expensive analytic calculation at the WAS (S_3); or
- Executing some lightweight pre-processing at the WAS (S_5), then having the DB fetch necessary data (S_6), and finally having the AS perform the remaining expensive analytic calculation (S_7).

The analytic process requires three different *service types*; namely, the WAS service type, the DB service type, and the AS service type. S_1, S_3, and S_5 are *instances* of the WAS service type since they are the services provided by the WAS. Similarly, S_2, S_4, and S_6 are instances of the DB service type, and S_7 is an instance of the AS service type.

Furthermore, there are three kinds of servers: WAS servers (M_1, M_2, and M_3); DB servers (M_4 and M_5); and AS servers (M_6). Although a server can typically

support any instance of its assigned service type, in general this is not always the case. Our example reflects this notion: each server is able to support all instances of its service type, except M_2 and M_4 are less powerful servers so that they cannot support computationally expensive service instances, S_3 and S_2.

Each server has a service level agreement (SLA) for each service instance it supports, and these SLAs are published and available for the scheduler. The SLA includes information such as a profile of the load versus response time and an upper bound on the request load size for which a server can provide a guarantee of its response time.

The scheduler is responsible for routing and coordinating execution of composite services comprising one or more implementations. A derived SLA can only be deployed with its corresponding routing logic. Note that the scheduler can derive SLA and routing logic as well as handle the task of routing the requests. Alternatively, the scheduler can be used solely for the purpose of deriving SLA and routing logic while configuring a content aware routers, such as [13], for high performance and hardware-based routing.

The scheduler can also be enhanced to perform the task of monitoring actual QoS achieved by workflow execution and by individual service providers. If the scheduler observes failure of certain service providers to their QoS published, it can re-compute feasible SLA and routing logic on demand to adapt to the runtime environment.

In this paper, we focus on the problem of automatically deriving the routing logic of a composite service with consideration of both *horizontal*and *vertical* load distribution options. The scheduler is required to find an optimal combination of a set of variables illustrated in Figure 3 for a number of concurrent requests. We discuss our scheduling approach next.

3 Scheduling Composite Services

3.1 Solution Space

In this section, we formally define the problem and describe how we model its complexity. We assume the following scenario elements:

- *Requests* for a workflow execution are submitted to a scheduling agent.
- The workflow can be embodied by one of several *implementations*, so each request is assigned to one of these implementations by the scheduling agent.
- Each implementation invokes several *service types*, such as a web application server, a DBMS, or a computational analytics server.
- Each service type can be embodied by one of several *instances* of the service type, where each instance can have different computing requirements. For example, one implementation may require heavy DBMS computation (such as through a stored procedure) and light computational analytics, whereas another imple-

mentation may require light DBMS querying and heavy computational analytics. We assume that these implementations are set up by administrators or engineers.
- Each service type is executed on a *server* within a pool of servers dedicated to that service type.

Each service type can be served by a pool of servers. We assume that the servers make agreements to guarantee a level of performance defined by the completion time for completing a web service invocation. Although these SLAs can be complex, in this paper we assume for simplicity that the guarantees can take the form of a linear performance degradation under load, an approach similar to other published work on service SLAs (e.g. [8]). This guarantee is defined by several parameters: α is the expected completion time (for example, on the order of seconds) if the assigned workload of web service requests is less than or equal to β, the maximum concurrency, and if the workload is higher than β, the expected completion for a workload of size ω is $\alpha + \gamma(\omega - \beta)$ where γ is a fractional coefficient. In our experiments we vary α, β, and γ with different distributions.

We would like to ideally perform optimal scheduling to simultaneously distribute the load both vertically (across different implementation options) and horizontally (across different servers supporting a particular service type). There are thus two stages of scheduling, as shown in Figure 4.

In the first stage, the requests are assigned to the implementations. In the second stage each implementation has a known set of instances of a service type, and each instance is assigned to servers within the pool of servers for the instance's service type. The solution space of possible scheduling assignments can be found by looking at the possible combinations of these assignments. Suppose there are R requests and M possible implementations. There are then M^R possible assignments in the first stage. Suppose further there are on average T service type invocations per implementation, and each of these service types can be handled by one of S on average possible servers. Across all the implementations, there are then S^T combinations of assignments in the second stage. It total, there are $M^R \cdot S^T$ combinations.

Clearly, an exhaustive search through this solution space is prohibitively costly for all but the smallest configurations. In the next subsection we describe how we use a genetic search algorithm to look for the optimal scheduling assignments.

3.2 Genetic algorithm

Given the solution space of $M^R \cdot S^T$, the goal is to find the best assignments of requests to implementations and service type instances to servers in order to minimize the running time of the workload, thereby providing our desired vertical and horizontal balancing. To search through the solution space, we use a genetic algorithm (GA) global search heuristic that allows us to explore portions of the space in a guided manner that converges towards the optimal solutions [12] [9]. We note that a GA is only one of many possible approaches for a search heuristic; others include

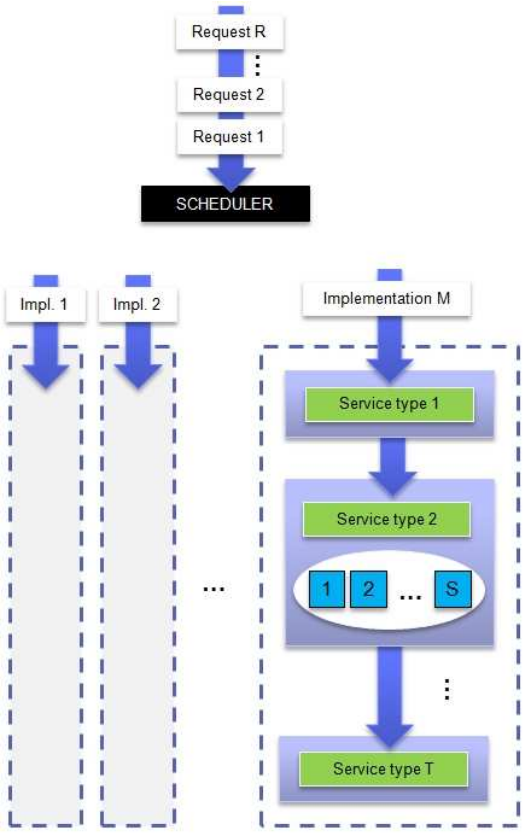

Fig. 4 The scheduling and assignment spans two stages. In the first stage, requests are assigned to implementations, and in the second stage, service type instances are assigned to servers.

tabu search, simulated annealing, and steepest-ascent hill climbing. We use a GA only as a tool.

A GA is a computer simulation of Darwinian natural selection that iterates through various generations to converge toward the best solution in the problem space. A potential solution to the problem exists as a chromosome, and in our case, a chromosome is a specific mapping of requests-to-implementations and instances-to-servers along with its associated workload execution time. Genetic algorithms are commonly used to find optimal exact solutions or near-optimal approximations in combinatorial search problems such as the one we address. It is known that a GA provides a very good tradeoff between exploration of the solution space and exploitation of discovered maxima [9]. Furthermore, a genetic algorithm does have an advantage of progressive optimization such that a solution is available at any time, and the result continues to improve as more time is given for optimization.

Algorithm 1 Genetic Search Algorithm

```
 1: FUNCTION Genetic algorithm
 2: BEGIN
 3: Time t
 4: Population P(t) := new random Population
 5:
 6: while ! done do
 7:    recombine and/or mutate P(t)
 8:    evaluate(P(t))
 9:    select the best P(t+1) from P(t)
10:    t := t + 1
11: end while
12: END
```

Note that the GA is not guaranteed to find the optimal solution since the recombination and mutation steps are stochastic.

Our choice of a genetic algorithm stemmed from our belief that other search heuristics (for example, simulated annealing) are already along the same lines as a GA. These are randomized global search heuristics, and genetic algorithms are a good representative of these approaches. Prior research has shown there is no clear winner among these heuristics, with each heuristic providing better performance and more accurate results under different scenarios [21, 16, 22]. Furthermore, from our own prior work, we are familiar with its operations and the factors that affect its performance and optimality convergence. Additionally, the mappings in our problem context are ideally suited to array and matrix representations, allowing us to use prior GA research that aid in chromosome recombination [6]. There are other algorithms that we could have considered, but scheduling and assignment algorithms are a research topic unto themselves, and there is a very wide of range of approaches that we would have been forced to omit.

Pseudo-code for a genetic algorithm is shown in Algorithm 1. The GA executes as follows. The GA produces an initial random population of chromosomes. The chromosomes then recombine (simulating sexual reproduction) to produce children using portions of both parents. Mutations in the children are produced with small probability to introduce traits that were not in either parent. The children with the best scores (in our case, the lowest workload execution times) are chosen for the next generation. The steps repeat for a fixed number of iterations, allowing the GA to converge toward the best chromosome. In the end it is hoped that the GA explores a large portion of the solution space. With each recombination, the most beneficial portion of a parent chromosome is ideally retained and passed from parent to child, so the best child in the last generation has the best mappings. To improve the GA's convergence, we implemented elitism, where the best chromosome found so far is guaranteed to exist in each generation.

3.2.1 Chromosome representation of a solution

We used two data structures in a chromosome to represent each of the two scheduling stages. In the first stage, R requests are assigned to M implementations, so its representative structure is simply an array of size R, where each element of the array is in the range of $[1, M]$, as shown in Figure 5.

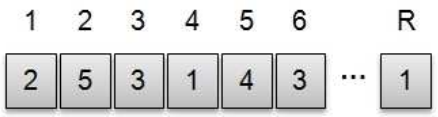

Fig. 5 An example chromosome representing the assignment of R requests to M implementations.

The second stage where instances are assigned to servers is more complex. In Figure 6 we show an example chromosome that encodes one scheduling assignment. The representation is a 2-dimensional matrix that maps {implementation, service type instance} to a service provider. For an implementation i utilizing service type instance j, the $(i, j)^{th}$ entry in the table is the identifier for the server to which the business process is assigned.

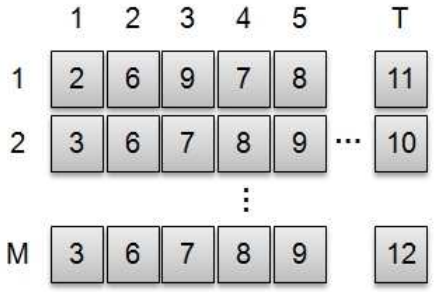

Fig. 6 An example chromosome representing a scheduling assignment of (implementation,service type instance) → service provider. Each row represents an implementation, and each column represents a service type instance. Here there are M workflows and T service types instances. In workflow 1, any request for service type 3 goes to server 9.

3.2.2 Chromosome recombination

Two parent chromosomes recombine to produce a new child chromosome. The hope is that the child contains the best contiguous chromosome regions from its parents.

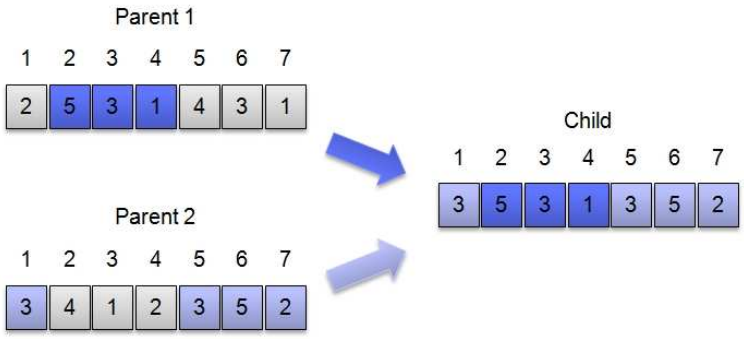

Fig. 7 An example recombination between two parents to produce a child for the first stage assignments. This recombination uses a 2-point crossover recombination of two one-dimensional arrays. Contiguous subsections of both parents are used to create the new child.

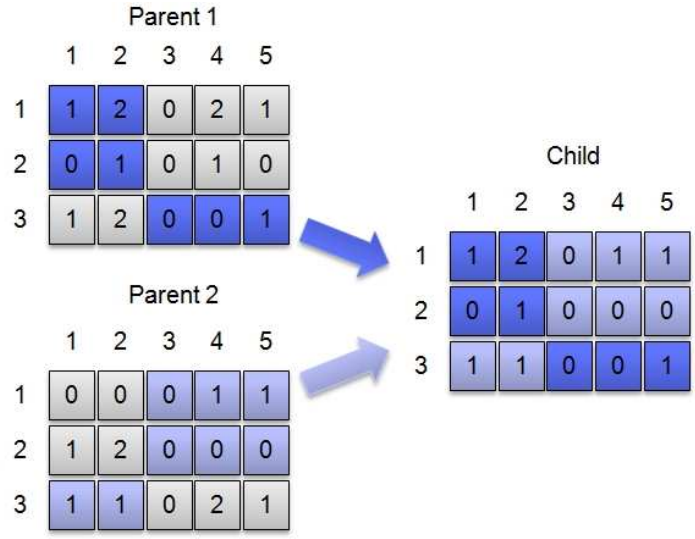

Fig. 8 An example recombination between two parents to produce a child for the second stage assignments. Elements from quadrants II and IV from the first parent and elements from quadrants I and III from the second parent are used to create the new child.

Recombining the chromosome from the first scheduling stage is simple since the chromsomes are simple 1-dimensional arrays. Two cut points are chosen randomly and applied to both the parents. The array elements between the cut points in the first parent are given to the child, and the array elements outside the cut points from the second parent are appended to the array elements in the child. This is known as a 2-point crossover and is shown in Figure 7.

For the 2-dimensional matrix, chromosome recombination was implemented by performing a one-point crossover scheme twice (once along each dimension). The crossover is best explained by analogy to Cartesian space as follows. A random location is chosen in the matrix to be coordinate (0, 0). Matrix elements from quadrants II and IV from the first parent and elements from quadrants I and III from the second parent are used to create the new child. This approach follows GA best practices by keeping contiguous chromosome segments together as they are transmitted from parent to child, as shown in Figure 8.

The uni-chromosome mutation scheme randomly changes one of the service provider assignments to another provider within the available range. Other recombination and mutation schemes are an area of research in the GA community, and we look to explore new operators in future work.

3.2.3 GA evaluation function

The evaluation function returns the resulting workload execution time given a chromosome. Note the function can be implemented to evaluate the workload in any way so long as it is consistently applied to all chromosomes across all generations.

Our evaluation function is shown in Algorithm 2. In lines 6 to 8, it initialises the execution times for all the servers in the chromosome. In lines 11-17, it assigns requests to implementations and service type instances to servers using the two mappings in the chromosome. The end result of this phase is that the instances are accordingly enqueued the servers. In lines 19-21 the running times of the servers are calculated. In lines 24-26, the results of the servers are used to compute the results of the implementations. The function returns the maximum execution time among the implementations.

3.3 Handling online arriving requests

As mentioned earlier, the problem domain we consider is that of batch-arrival request routing. We take full advantage of such a scenario through the use of the GA, which has knowledge of the request population. We can further extend this approach to online arriving requests, a lengthy discussion which we omit here due to space limits. A typical approach is to aggregate the incoming requests into a queue, and when a designated timer expires, all requests in the queue at that time are scheduled. There may still be uncompleted requests from the previous execution, so the requests may be mingled together to produce a larger schedule. An alternative approach is to use online stochastic optimization techniques commonly found in online decision-making systems [11].

First, we can continue to use the GA, but instead of having the complete collection of requests available to us, we can allow requests to aggregate into a queue first. When a periodic timer expires, we can run the GA on those requests while ag-

Algorithm 2 GA evaluation function

```
 1: FUNCTION evaluate
 2: IN: CHROMOSOME, a representation of the assignments of requests to implementation and
    service type instances to servers
 3: OUT: runningtime, the running time of this workload
 4: BEGIN
 5:
 6: for (each server ∈ CHROMOSOME) do
 7:    set server's running time to 0
 8: end for
 9:
10: {Loop over each request and its implementations}
11: for (each request ∈ CHROMOSOME) do
12:    implementation := requests's implementation
13:    for (each instance ∈ implementation) do
14:       server := implementation's server
15:       Enqueue this job at server
16:    end for
17: end for
18:
19: for (each server) do
20:    Compute the running time of server
21: end for
22:
23: {Now compute the running time of the implementations}
24: for (each implementation ∈ CHROMOSOME) do
25:    Aggregate the running time of this implementation across its instances
26: end for
27:
28: runningtime := maximum running time of each implementation
29: return runningtime
30: END
```

gregating any more incoming requests into another queue. Once the GA is finished with the first queue, it will process the next queue when the periodic timer expires again. If the request arrival rate is faster than the GA's processing rate, we can take advantage of the fact that the GA can be run as an incomplete, near-optimal search heuristic: we can go ahead and let the timer interrupt the GA, and the GA will have *some* solutions that, although sub-optimal, is probabilistically better than a greedy solution. This typical methodology is also shown in [20], where requests for broadcast messages are queued, and the messages are optimally distributed through the use of an evolutionary strategies algorithm (a close cousin of a genetic algorithm).

Second (and unrelated to genetic algorithms), we can use online stochastic optimization techniques to serve online arrivals. This approach approximates the offline problem by sampling historical arrival data in order to make the best online decision. A good overview is provided in [19]. In this technique, the online optimizer receives an incoming sequence of requests, gets historical data over some period of time from a sampling function that creates a statistical distribution model, and then

calculates and returns an optimized allocation of requests to available resources. This optimization can be done on a periodic or continuous basis.

4 Experiments and Results

We ran experiments to show how our system compared to other well-known algorithms with respect to our goal of providing request routing with horizontal and vertical distribution. Since one of our intentions was to demonstrate how our system scales well up to 1000 requests, we used a synthetic workload that allowed us to precisely control experimental parameters, including the number of available implementations, the number of published service types, the number of service type instances per implementation, and the number of servers per service type instance. The scheduling and execution of this workload was simulated using a program we implemented in standard C++. The simulation ran on an off-the-shelf Red Hat Linux desktop with a 3.0 GHz Pentinum IV and 2GB of RAM.

In these experiments we compared our algorithm against the following alternatives:

- A *round-robin* algorithm that assigns requests to an implementation and service type instances to a server in circular fashion. This well-known approach provides a fast and simple scheme for load-balancing.
- A *random-proportional* algorithm that proportionally assigns instances to the servers. For a given service type, the servers are ranked by their guaranteed completion time, and instances are assigned proportionally to the servers based on the servers' completion time. (We also tried a proportionality scheme based on both the completion times and maximum concurrency but attained the same results, so only the former scheme's results are shown here.) To isolate the behavior of this proportionality scheme in the second phase of the scheduling, we always assigned the requests to the implementations in the first phase using a round-robin scheme.
- A *purely random* algorithm that randomly assigns requests to an implementation and service type instances to a server in random fashion. Each choice was made with a uniform random distribution.
- A *greedy* algorithm that always assigns business processes to the service provider that has the fastest guaranteed completion time. This algorithm represents a naive approach based on greedy, local observations of each workflow without taking into consideration all workflows.

In the experiments that follow, all results were averaged across 20 trials, and to help normalize the effects of any randomization used during any of the algorithms, each trial started by reading in pre-initialized data from disk. In Table 1 we list our experimental parameters for our baseline experiments. We vary these parameters in other experiments, as we discuss later.

Experimental parameter	Comment
Requests	1 to 1000
Implementations	5, 10, 20
Service types used per implementation	uniform random: 1 - 10
Instances per service type	uniform random: 1 - 10
Servers per service type	uniform random: 1 - 10
Server completion time (α)	uniform random: 1 - 12 seconds
Server maximum concurrency (β)	uniform random: 1 - 12
Server degradation coefficient (γ)	uniform random: 0.1 - 0.9
GA: population size	100
GA: number of generations	200

Table 1 Experiment parameters.

4.1 Baseline configuration results

In Figures 9, 10, and 11 we show the behavior of the algorithms as they schedule requests against 5, 10, and 20 implementations, respectively. In each graph, the x-axis shows the number of requests (up to 1000), and the y-axis is average response time upon completing the workload. This response time is the *makespan*, the metric commonly used in the scheduling community and calculated as the maximum completion time across all requests in the workload. As the total number of implementations increases across the three graphs, the total number of service types, instances, and servers scaled as well in accordance to the distributions of these variables from Table 1. In each of the figures, it can be see that the GA is able to produce a better assignment of requests to implementations and service type instances to servers than the other algorithms. The GA shows a 45% improvement over its nearest competitor (typically the round-robin algorithm) with a configuration of 5 implementations and 1000 requests and a 36% improvement in the largest configuration with 20 implementations and 1000 requests.

The relative behavior of the other algorithms was consistent. The greedy algorithm performed the worst while the random-proportional and random algorithms were close together. The round-robin came the closest to the GA.

To better understand these results, we looked at the individual behavior of the servers after the instance requests were assigned to them. In Figure 12 we show the percentage of servers that were saturated among the servers that were actually assigned instance requests. These results were from the same 10-implementation experiment from Figure 10. For clarity, we focus on a region with up to 300 requests.

We consider a server to be saturated if it was given more requests than its maximum concurrency parameter. From this graph we see the key behavior that the GA is able to find assignments well enough to delay the onset of saturation until 300 requests. The greedy algorithm, as can be expected, always targets the best server from the pool available for a given service type and quickly causes these chosen servers to saturate. The round robin is known to be a quick and easy way to spread load and indeed provides the lowest saturation up through 60 requests. The

random-proportional and random algorithms reach saturation points between that of the greedy and GA algorithms.

4.2 Effect of service types

We then varied the number of service types per implementation, modeling a scenario where there is a heavily skewed number of different web services available to each of the alternative implementations. Intuitively, in a deployment where there is a large number of services types to be invoked, the running time of the overall workload will increase.

In Figure 13 we show the results where we chose the numbers of service types per implementation from a Gaussian distribution with a mean of 2.0 service types; this distribution is in contrast to the previous experiments where the number was selected from a uniform distribution in the inclusive range of 1 to 10. As can be seen, the algorithms show the same relative performance from prior results in that the GA is able to find the scheduling assignments resulting in the lowest response times. The worst performer in this case is the random algorithm. In Figure 14 we skew the number of service types in the other direction with a Gaussian distribution with a mean of 8.0. In this case the overall response time increases for all algorithms, as can be expected. The GA still provides the best response time.

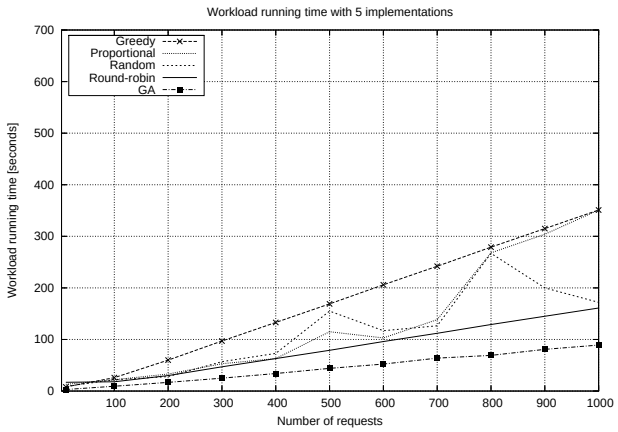

Fig. 9 Response time with 5 implementations.

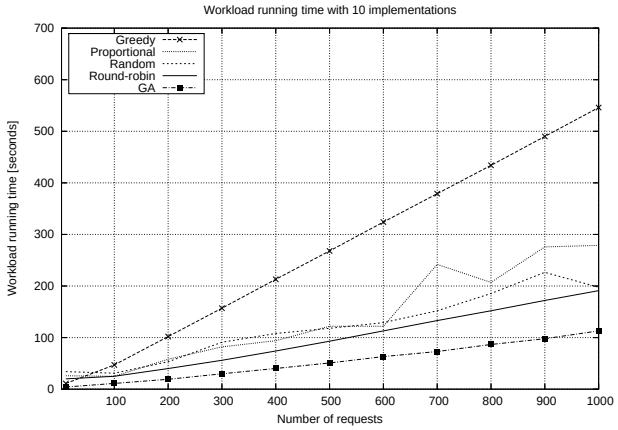

Fig. 10 Response time with 10 implementations.

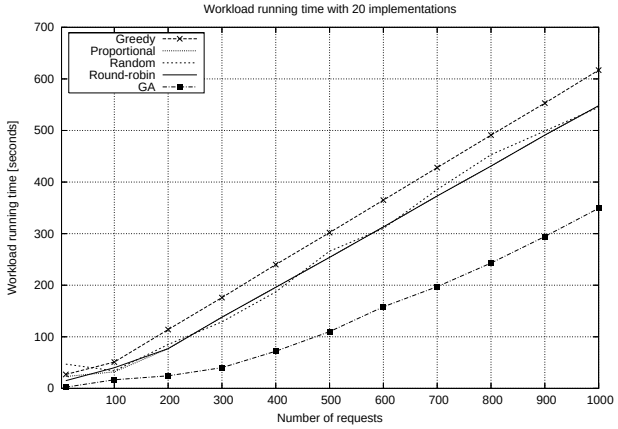

Fig. 11 Response time with 20 implementations.

4.3 Effect of service type instances

In these experiments we varied the number of instances per service type. We implemented a scheme where each instance incurs a different running time on each server; that is, a unique combination of instance and server provides a different response time, which we put into effect by a Gaussian random number generator. This approach models our target scenario where a given implementation may run an instances that performs more or less of the work associated with the instance's service type. For example, although two implementations may require the use of a DBMS, one implementation's instance of this DBMS task may require less compu-

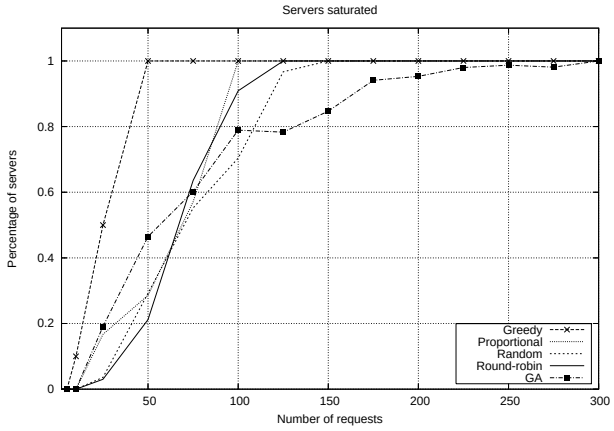

Fig. 12 Percentage of servers that were saturated. A saturated server is one whose workload is greater than its maximum concurrency.

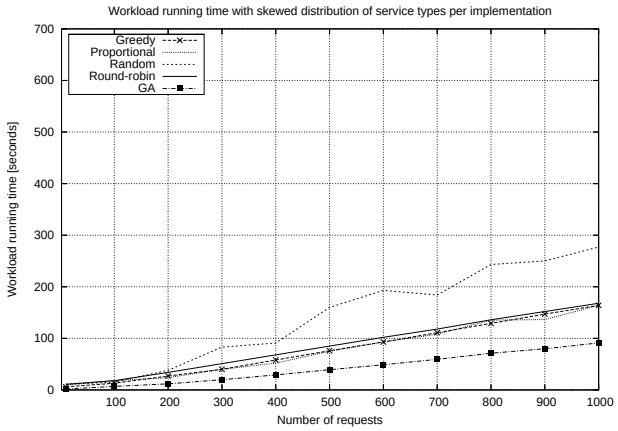

Fig. 13 Average response time with a skewed distribution of service types per implementation. The distribution was Gaussian (λ = 2.0, σ = 2.0 service types).

tation than the other implementation due to the offload of a stored procedure in the DBMS to a separate analytics server. Our expectation is that having more instances per service type allows a greater variability in performance per service type.

Figure 15 shows the algorithm results when we skewed the number of instances per service type with a Gaussian distribution with a mean of 2.0 instances. Again, the relative ordering shows that the GA is able to provide the lowest workload response among the algorithms throughout. When we weight the number of instances with a mean of 8.0 instances per service type, as shown in Figure 16, we can see that the the GA again provides the lowest response time results. In this larger configuration,

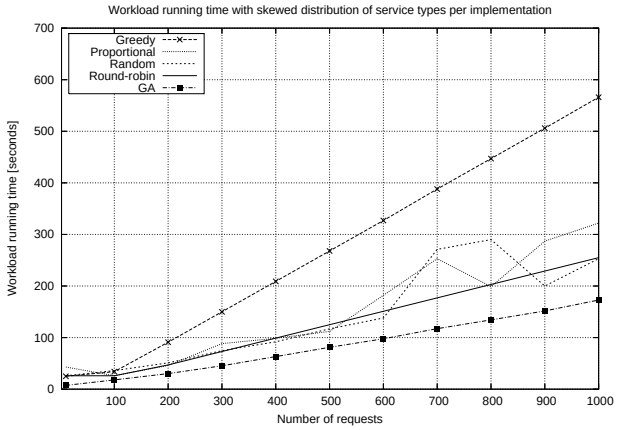

Fig. 14 Average response time with a skewed distribution of service types per implementation. The distribution was Gaussian (λ = 8.0, σ = 2.0 service types).

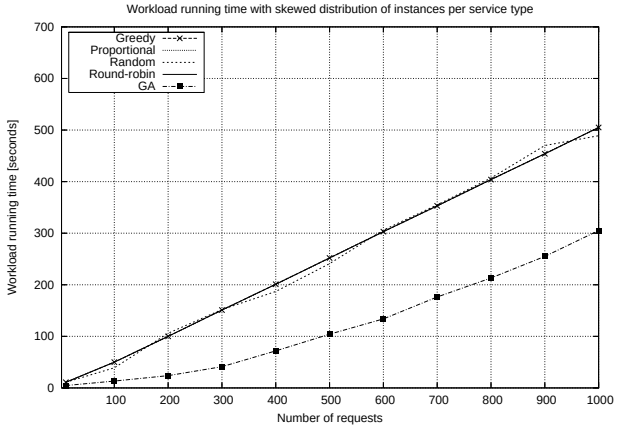

Fig. 15 Average response time with a skewed distribution of instances per service type. The distribution was Gaussian (λ = 2.0, σ = 2.0 instances).

the separation between all the algorithms is more evident with the greedy algorithm typically performing the worst; its behavior is again due the fact that it assigns jobs only to the best server among the pool of servers for a service type.

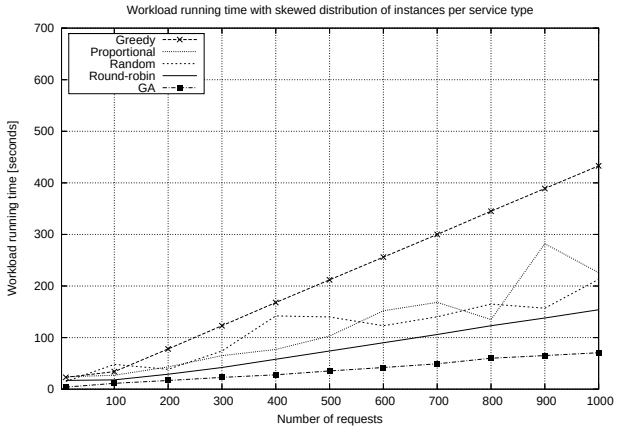

Fig. 16 Average response time with a skewed distribution of instances per service type. The distribution was Gaussian (λ = 8.0, σ = 2.0 instances).

4.4 Effect of servers (horizontal balancing)

Here we explored the impact of having more servers available in the pool of servers for the service types. This experiment isolates the effect of horizontal balancing. Increasing the size of this pool will allow assigned requests to be spread out and thus reduce the number of requests per server, resulting in lower response times for the workload. In Figures 17 and 18 we show the results with Gaussian distributions with means of 2.0 and 8.0, respectively. In both graphs the GA appears to provide the lowest response times. Furthermore, it is interesting to note that in the random, random-proportional, and round-robin algorithms, the results did not change substantially between the two experiments even though the latter experiment contains four times the average number of servers. We believe this result may be due to the fact that the first-stage scheduling of requests to implementations is not taking sufficient advantage of the second-stage scheduling of service type instances to the increased number of servers. Since the GA is able to better explore all combinations across both scheduling stages, it is able to produces its better results. We will explore this aspect in more detail in the future.

4.5 Effect of server performance

In this subsection we look at the impact on the servers' individual performance on the overall workload running time. In previous sections we described how we modeled each server with variables for the response time (α) and the concurrency

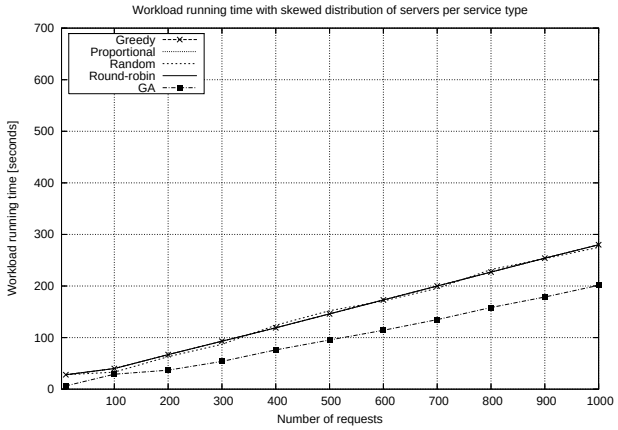

Fig. 17 Average response time with a skewed distribution of servers per service type. The distribution was Gaussian (λ = 2.0, σ = 2.0 instances).

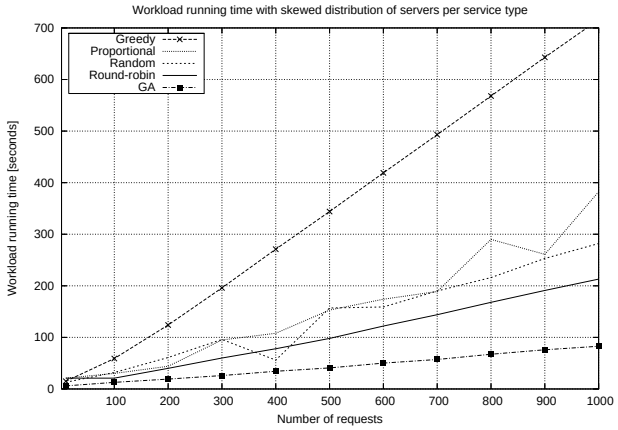

Fig. 18 Average response time with a skewed distribution of servers per service type. The distribution was Gaussian (λ = 8.0, σ = 2.0 instances).

(β. Here we skewed these variables to show how the algorithms performed as a result.

In Figures 19 and 20 we skewed the completion times with Gaussian distributions with means of 2.0 and 9.0, respectively. It can be seen that the relative orderings of the algorithms are roughly the same in each, with the GA providing best performance, the greedy algorithm giving the worst, and the other algorithms running in between. Surprisingly, the difference in response time between the two experiments was much less than we expected, although there is a slight increase in all the algorithms except for the GA. We believe that the lack of a dramatic rise in overall

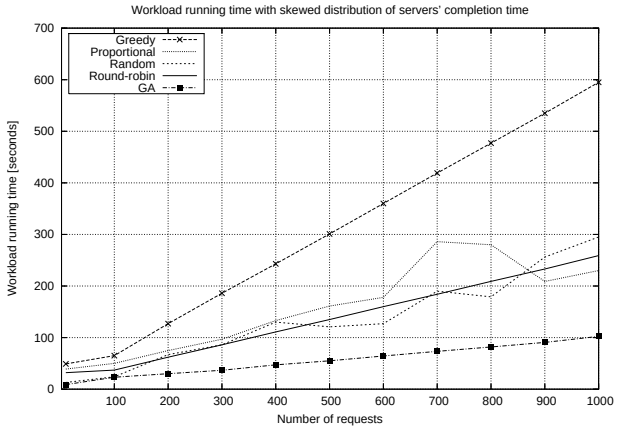

Fig. 19 Average response time with a skewed distribution of servers' completion time. The distribution was Gaussian (λ = 2.0, σ = 2.0 seconds).

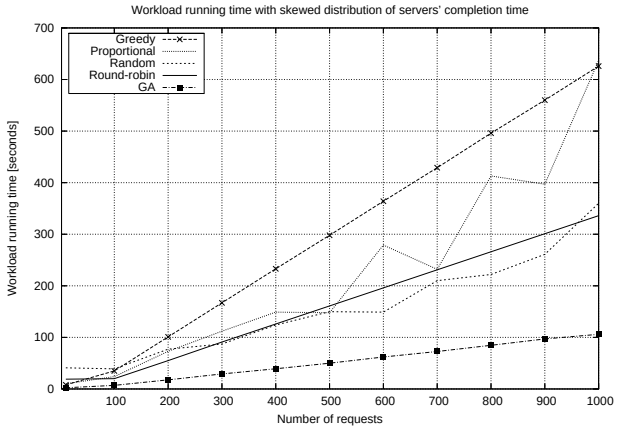

Fig. 20 Average response time with a skewed distribution of servers' completion time. The distribution was Gaussian (λ = 9.0, σ = 2.0 seconds).

response time is due to whatever load balancing is being performed by the algorithms (except the greedy algorithm).

We then varied the maximum concurrency variable for the servers using Gaussian distributions with means of 2.0 and 9.0, as shown in Figures 21 and 22. From these results it can be observed that the algorithms react well with an increasing degree of maximum concurrency. As more requests are being assigned to the servers, the servers respond with faster response times when they are given more headroom to run with these higher concurrency limits.

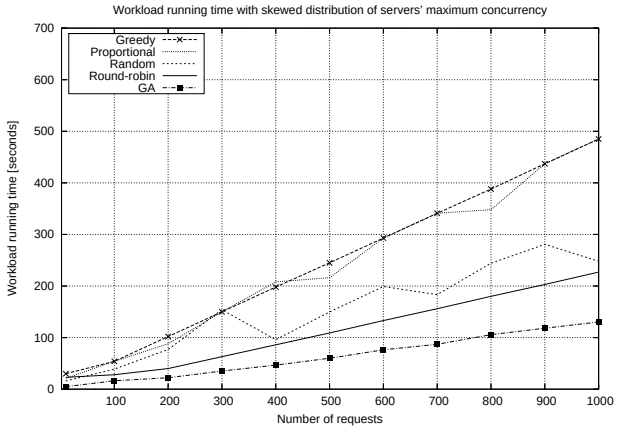

Fig. 21 Average response time with a skewed distribution of servers' maximum concurrency. The distribution was Gaussian (λ = 4.0, σ = 2.0 jobs).

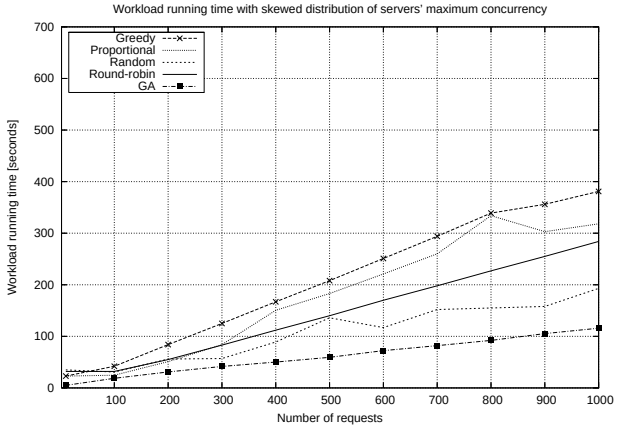

Fig. 22 Average response time with a skewed distribution of servers' maximum concurrency. The distribution was Gaussian (λ = 11.0, σ = 2.0 jobs).

4.6 Effect of response variation control

We additonally evaluated the effect of having the GA minimize the variation in the requests' completion time. As mentioned earlier, we have been been calculating the workload completion as the maximum completion time of the requests in that workload. While this approach has been effective, it produces wide variation between the requests' completion times due to the stochastic packing of requests by the GA. This variation in response time, known as *jitter* in the computer networking community, may not be desirable, so we further provided an alternative objective function

that minimizes the jitter (rather than minimizing the workload completion time). In Figure 23 we show the average standard deviations resulting from these different objective functions (using the same parameters as in Figure 10). With variation minimization on, the average standard deviation is always close to 0, and with variation minimization off, we observe an increasing degree of variation. The results in Figure 24 show that the reduced variation comes at the cost of longer response times.

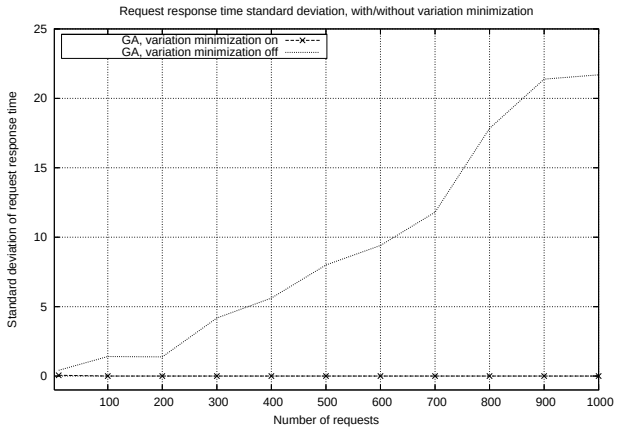

Fig. 23 Average standard deviation from the mean response for two different objective functions.

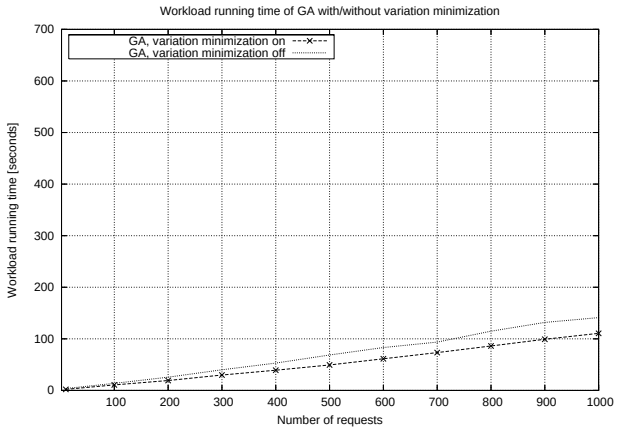

Fig. 24 Average response time for two different objective functions.

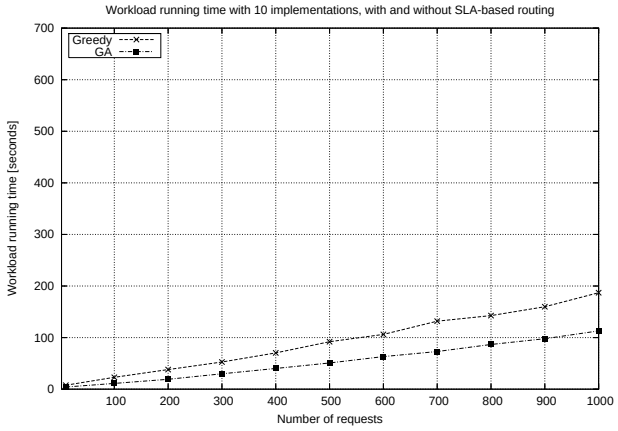

Fig. 25 Response time with 10 implementations with configurations for SLA and without SLA.

4.7 Effect of routing against conservative SLA

We looked at the GA behavior when its input parameters were not the servers' actual parameters but rather the parameters provided by a conservative SLA. In some systems, SLAs may be defined with a safety margin in mind so that clients of the service do not approach the actual physical limits of the underlying service. In that vein, we ran an experiment similar to that shown in Figure 10, but in this configuration we used parameters for the underlying servers with twice the expected response time and half the available parallelism, mirroring a possible conservative SLA. As can be seen in Figure 25, the GA converges towards a scheduling where the extra slack given by the conservative SLA results in a slower response time.

4.8 Summary of experiments

In this section we evaluated our GA reference implementation of a scheduler that performs request-routing for horizontal and vertical load distribution. We showed that the GA consistently produces lower workload response time than its competitors. Furthermore, as can be expected, the scheduler is sensitive to a number of parameters, including the number of service types in each implementation, the number of service type instances, the number of servers, the per-server performance, the desired degree of variation, and the tightness of the SLA parameters.

5 Related Work

[23] described a distributed quality of service (QoS) management architecture and middleware that accommodates and manages different dimensions and measures of QoS. The middleware supports the specification, maintenance and adaptation of end-to-end QoS (including temporal requirements) provided by the individual components in complex real time application systems. Using QoS negotiation, the middleware determines the quality levels and resource allocations of the application components. This work focused on analysis tradeoff between QoS and cost instead of ensuring QoS requirements in our paper.

[30] presented two algorithms for finding replacement services in autonomic distributed business processes when web service providers fail to response or meet the QoS requirement: following alternative predefined routes or finding alternative routes on demand. The algorithms give the QoS brokerage service fault tolerance capability and is complementary to our work.

In [31, 32, 33], Yu et. al developed a set of algorithms for Web services selection with end-to-end QoS constraints. A key difference between our work and theirs is that they simplify and reduce the complexity space considerably, something which we do not do. They take all incoming workflows, aggregate them into one singe workflow, and then schedule that one workflow onto the underlying service providers. We do not do this aggregation, and therefore our approach provides a higher degree of scheduling flexibility.

Fig. 26 Aggregation of Service Workflows

Consider the two workflows shown on the left of Figure 26 where each task in the workflow invokes a particular service type. In their work, they aggregate the workflows into a single function graph, resulting in a simplified form shown on the right of Figure 26.

Each service type is then mapped onto a service provider chosen from the pool of service providers for that type. It is important to note that each service type is assigned to the same chosen provider, even though the instances of that service type are different. For example, because both *workflow 1* and *workflow 2* use *S*3, both instances are mapped to the same provider.

In our work, we do not do this aggregation to reduce the complexity space. We consider unique combinations of {workflow, service type} and map these to a service provider. Thus, in our work, *S*3 in *workflow 1* may map to a different provider than *S*3 in *workflow 2*. This distinction allows for more flexible scheduling and potentially better turnaround time than their work.

In the work [27], the GA algorithm was used for load distribution for database cluster. In this work, the analytic workloads are distributed across a database cluster. The load distribution algorithm needs to consider collocation of MQTs (i.e. materialized views) with queries which can utilize them to improve performance, collocation of MQTs and the base tables which are needed to construct the MQTs, and minimizing the execution time of the whole workload on the database cluster. This work is a kind of *horizontal* load distribution. Similarly, the GA algorithm is also used in [26] to schedule query execution and view materialization sequence for minimal overall execution time.

Our work is related to prior efforts in web service composition, web service scheduling, and job scheduling. A web service's interface is expressed in WSDL, and given a set of web services, a workflow can be specified in a flow language such as BPEL4WS [1] or WSCI [15]. Several research projects have looked to provide automated web services composition using high-level rules (e.g. eFlow [4], SWORD [18]). Our work is complementary to this area, as we schedule business processes within multiple, already-defined workflows to the underlying service providers.

In the context of service assignment and scheduling, [34] maps web service calls to potential servers but their work is concerned with mapping only single workflows; our principal focus is on scalably scheduling multiple workflows (up to one thousand). [28] presents a dynamic provisioning approach that uses both predictive and reactive techniques for multi-tiered Internet application delivery. However, the provisioning techniques do not consider the challenges faced when there are alternative query execution plans and replicated data sources. [24] presents a feedback-based scheduling mechanism for multi-tiered systems with back-end databases, but unlike our work, it assumes a tighter coupling between the system components.

The work in [14] creates end-to-end paths for services (such as transcoding) and assigns servers on a hop-by-hop basis by minimising network latency between hops. Our work is complementary in that service assignment is based on business value metrics defined by agreed-upon service level agreements.

An SLA can be complex, requiring IT staff to translate from the legal document level description to system-specific requirement for deployment and enforcement. [29] proposed a framework for configuring extensible SLA management systems. In this work, an SLA is represented in XML format. In [3], an SLA execution manager (SAM) is proposed to manage cross-SLA execution that may involve an SLA with different terms. The work provides metadata management functionality for SLA aware scheduling presented in this paper. Thus, it is complementary to our work.

[25, 10] applied peer-to-peer technology for support real time services, such as data dissemination across internet with QoS assurance. In their context, they create an application-layer network route across multiple service nodes in order to provide some end-to-end service. This routing occurs in two steps: the user's high-level request is mapped to a service template, and then the template is mapped to a route of servers. This approach is similar to ours in that our business processes request service from the service types, and the service types must instantiated by assigning the business processes to an underlying server. The key differences are that: (1) their work is constrained by the topology of the application-layer network. Their work

looks at pipelines of service nodes in a line. The problem is finding routes through a network by adapting Dijkstra's algorithm for finding shortest path whereas our problem is assigning business processes to servers; Their work looks at pipelines of service nodes in a line; whereas our work looks at a more flexible workflow condition that may involve branches, including AND and OR; (3) their primary metrics are availability and latency, whereas we use a more flexible and generalizable business value to evaluate assignments. Furthermore, our work supports an infrastructure where a server can support multiple service types (c.f. our scenario is that business processes within a workflow must be scheduled onto web service providers). The salient differences are that the machines can process only one job at a time (we assume servers can multi-task but with degraded performance and a maximum concurrency level), tasks within a job cannot simultaneously run on different machines (we assume business processes can be assigned to any available server), and the principal metric of performance is the *makespan*, which is the time for the last task among all the jobs to complete. As we showed, optimizing on the makespan is insufficient for scheduling the business processes, necessitating different metrics.

6 Conclusion

Cloud computing aims to do the dirty work for the user: by moving issues of mangement and provisioning away from the end consumer and into the server-side data centers, users are given more freedom to pick and choose the applications that suit their needs. However, computing in the clouds depends heavily on the scalablity and robustness of the underlying cloud architecture.

We discussed enterprise cloud computing where enterprises may use a service-oriented architecture to publish a streamlined interface to their business processes. In order to scale up the number of business processes, each tier in the provider's architecture usually deploys multiple servers for load distribution and fault tolerance. Such load distribution across multiple servers within the same tier can be viewed as *horizontal* load distribution. One limitation of this approach is that load cannot be distributed further when all servers in the same tier are fully loaded. Another approach for providing resiliency and scalabilty is to have *multiple implementation options* that give opportunities for *vertical* load distribution across tiers.

We described in detail a request routing framework for SOA-based enterprise cloud computing that takes into account both these options for *horizontal* and *vertical* load distribution. Experiments showed that our algorithm and methodology can scale well up to a large-scale system configuration comprising up to 1000 workflow requests directed to a complex composite service with multiple implementation options available. The experimental results also demonstrate that our framework is more agile in the sense that it is effective in dealing with mis-configured infrastructures in which there are too many or too few servers in one tier. As a result, our framework can effectively utilize available multiple implementations to distribute loads across tiers.

References

1. Business process execution language for web services, v 1.1, 2005. www-128.ibm.com/developerworks/library/ws-bpel/.
2. Cloud computing: Clash of the clouds. *The Economist*, 2009.
3. Melissa J. Buco, Rong N. Chang, Laura Z. Luan, Christopher Ward, Joel L. Wolf, Philip S. Yu, Tevfik Kosar, and Syed Umair Ahmed Shah. Managing ebusiness on demand sla contracts in business terms using the cross-sla execution manager sam. In *ISADS*, pages 157–, 2003.
4. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of CAISE*, 2000.
5. Cisco. Ace application-level load balancer.
6. L. Davis. Job Shop Scheduling with Genetic Algorithms,. In *Proceedings of the International Conference on Genetic Algorithms*, 1985.
7. M. Dikaiakos, G. Pallis, D. Katsaros, P. Mehra, and A. Vakali. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet Computing*, 13(5):10–13, 2009.
8. G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
9. D. Goldberg. *Genetic Algorithms in Searth, Optimization, and Machine Learning*. Kluwer Academic, 1989.
10. Xiaohui Gu, Klara Nahrstedt, Rong N. Chang, and Christopher Ward. Qos-assured service composition in managed service overlay networks. In *ICDCS*, pages 194–, 2003.
11. Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization*. MIT Press, 2006.
12. J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.
13. Cisco System Inc. Scalable content switch.
14. J. Jin and K. Nahrstedt. On Exploring Performance Optimisations in Web Service Composition. In *Proceedings of Middleware*, 2004.
15. J. Josephraj. Web Services Choreography in Practice. In *www-128.ibm.com/developerworks/library/ws-choreography*, 2007.
16. L. Costa and P. Oliveira. Evolutionary algorithms approach to the solution of mixed integer nonlinear programming problems. In *Comput. Chem. Eng. 25*, 2001.
17. F5 Networks. Big-ip application-level load balancer.
18. S. Ponnekanti and A. Fox. Interoperability among Independently Evolving Web Services. In *Proceedings of Middleware*, 2004.
19. R. Bent and P. Van Hentenryck. Regrets Only! Online Stochastic Optimization Under Time Constraints. In *Nineteenth National Conference on Artificial Intelligence*, 2004.
20. R. Dewri and I. Ray and I. Ray and D. Whitley. Optimizing On-Demand Data Broadcast Scheduling in Pervasive Environments. 2008.
21. R. Lima and G. Francois and B. Srinivasan and R. Salcedo. Dynamic optimization of batch emulsion polymerization using MSIMPSA, a simulated-annealing-based algorithm. *Ind. Eng. Chem. Res.*, 43(24), 2004.
22. R. Oliveira and R. Salcedo. Benchmark testing of simulated annealing, adaptive random search and genetic algorithms for the global optimization of bioprocesses. In *International Conference on Adaptive and Natural Computing Algorithms*, 2005.
23. Mallikarjun Shankar, Miguel De Miguel, and Jane W.-S. Liu. An end-to-end qos management architecture. In *Proceedings of the Fifth IEEE Real Time Technology and Applications Symposium*.
24. G. Soundararajan, K. Manassiev, J. Chen, A. Goel, and C. Amza. Back-end Databases in Shared Dynamic Content Server Clusters. In *Proceedings of ICAC*, 2005.
25. Chunqiang Tang, Rong N. Chang, and Edward So. A distributed service management infrastructure for enterprise data centers based on peer-to-peer technology. In *IEEE SCC*, pages 52–59, 2006.
26. Thomas Phan and Wen-Syan Li. Dynamic Materialization of Query Views for Data Warehouse Workloads. In *Proceedings of the International Conference on Data Engineering*, 2008.

27. Thomas Phan and Wen-Syan Li. Load distribution of analytical query workloads for database cluster architectures. 2008.
28. B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi-Tier Internet Applications. In *Proceedings of ICAC*, 2005.
29. Christopher Ward, Melissa J. Buco, Rong N. Chang, Laura Z. Luan, Edward So, and Chunqiang Tang. Fresco: A web services based framework for configuring extensible sla management systems. In *ICWS*, pages 237–245, 2005.
30. Tao Yu and Kwei-Jay Lin. Adaptive algorithms for finding replacement services in autonomic distributed business processes. In *Proc. of the 7th International Symposium on Autonomous Decentralized Systems*, Chengdu, China, 2005.
31. Tao Yu and Kwei-Jay Lin. Service selection algorithms for web services with end-to-end qos constraints. *Inf. Syst. E-Business Management*, 3(2):103–126, 2005.
32. Tao Yu and Kwei-Jay Lin. Qcws: An implementation of qos-capable multimedia web services. *Multimedia Tools and Applications*, 30(2):165–187, 2006.
33. Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1), 2007.
34. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng. Quality Driven Web Services Composition. In *Proceedings of WWW*, 2003.