Kara et al. Journal of Cloud Computing: Advances, Systems and Applications 2014, 3:15
http://www.journalofcloudcomputing.com/content/3/1/15

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Genetic-based algorithms for resource management in virtualized IVR applications

Nadjia Kara[1*], Mbarka Soualhia[1], Fatna Belqasmi[2], Christian Azar[2] and Roch Glitho[2]

Abstract

Interactive Voice Response (IVR) is a technology that allows automatic human-computer interactions, via a telephone keypad or voice commands. The systems are widely used in many industries, including telecommunications and banking. Virtualization is a potential technology that can enable the easy development of IVR applications and their deployment on the cloud. IVR virtualization will enable efficient resource usage by allowing IVR applications to share different IVR substrate components such as the key detector, the voice recorder and the dialog manager. Resource management is part and parcel of IVR virtualization and poses a challenge in virtualized environments where both processing and network constraints must be considered. Considering several objectives to optimize the resource usage makes it even more challenging. This paper proposes IVR virtualization task scheduling and computational resource sharing (among different IVR applications) strategies based on genetic algorithms, in which different objectives are optimized. The algorithms used by both strategies are simulated and the performance measured and analyzed.

Keywords: Resource management; Cloud computing; Virtualization; IVR applications; Genetic algorithms

Introduction

Interactive Voice Response (IVR) is a technology that allows automatic human-computer interactions, via a telephone keypad or voice commands. Its key function is to provide end-users with self-service voice information [1]. IVR systems are widely used in many industries, including telecommunications and banking, to improve customer satisfaction, reduce cost, and ensure uninterrupted service. Examples of IVR applications include automated attendants, automated meter readers and IVR banking. The automated attendant transfers callers to the appropriate extensions automatically, without intervention by a receptionist; using automated meter readers, utilities customers can remotely enter their meter readings, while IVR banking allows end-users to consult their bank balance or last transactions, for instance.

Virtualization is a potential technology that can enable the easy development of IVR applications and their deployment on the cloud. It allows the abstraction and sharing of computer and network resources, as well as the co-existence of entities on the same substrates [2]. Cloud computing is a multi-facet paradigm, which enables the easy introduction of new services, scalability and efficient resource usage. The main facets of cloud computing are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [3]. IaaS provides the pool of virtualized resources that are used by applications provisioned (to end-users or other applications) as SaaS. The development and management of such applications are made easier through PaaS which adds one or more levels of abstraction to the infrastructures offered by IaaS providers.

IVR virtualization will enable efficient resource usage by allowing IVR applications to share different IVR substrate components such as key detectors, voice recorders and dialog managers. It will also ease the development and the management of IVR applications that can be offered as cloud-based services.

In a previous work [4,5], we proposed a virtualized infrastructure for IVR applications in the cloud. The proposed architecture is composed of three layers (substrate, infrastructure, and platform) and an IVR substrate repository (Figure 1). The substrate layer provides IVR substrates that can be composed and assembled on the fly to build IVR applications. These substrates are accessible via the

* Correspondence: Nadjia.kara@etsmtl.ca
[1]ETS, University of Quebec, 1100, Notre-Dame street West, Montreal, Quebec H3C 1K3, Canada
Full list of author information is available at the end of the article

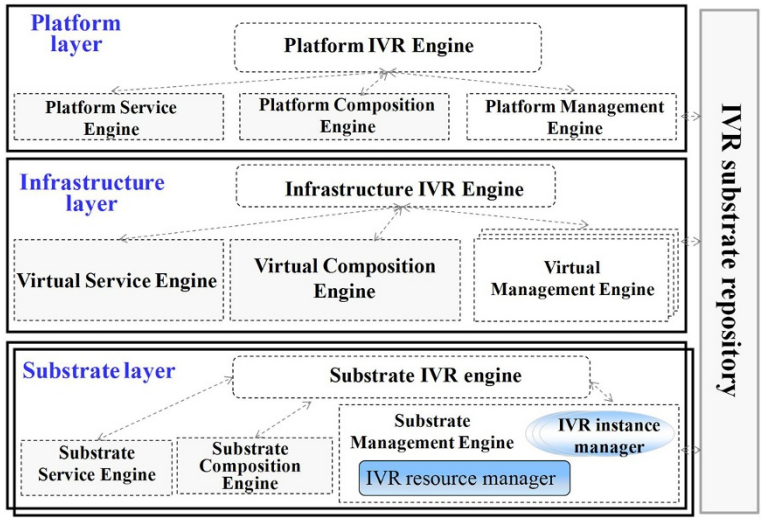

Figure 1 Virtualized infrastructure for IVR applications.

infrastructure layer. The platform layer for its part adds one or more abstractions and makes the substrates available to the IVR applications' developers while the IVR substrate repository is used to publish and discover existing IVR substrates.

The three layers communicate via three planes: service, composition and management. The service plane handles the service execution, including coordinating the execution of services that involve several substrates; the composition plane intervenes in the composition of the appropriate substrates to create a given IVR application and the management plane is responsible for the actual control and management of substrate resources. It allows the instantiation of IVR applications and related substrates, enables fault and performance monitoring, and performs accounting for charging purposes. Each layer includes one functional entity at each plane, and one key entity that coordinates the operation of the layer entities at the three planes. At the substrate layer for instance we have a substrate service, composition and management engines that are coordinated by the substrate IVR engine.

This paper focuses on the management plane, and more precisely on resource management at the substrate layer. Before a service provider can make an IVR application available to its end-users, he should develop such an application by discovering and (eventually) composing existing substrates. It then activates the application, a phase which includes the instantiation of the substrates required to run the application. It is only after this that the end-users can interact with the application.

To instantiate new substrates, the substrate management engine should identify the resources needed, verify resource availability and then allocate the appropriate resources. We focus on two issues: computational resources sharing and task scheduling. In computational resource sharing, we examine the sharing of existing computational resources (e.g., virtual machines, processors) between different IVR applications optimally. Task sch\eduling relates to the assignment of the instantiation requests received.

Computational resource sharing and task scheduling in virtualized environments where both processing and network constraints must be considered are challenging issues. Considering several objectives to optimize the resource scheduling and usage makes them even more challenging.

Several research studies have focused on load balancing across processors or computers in both non-virtualized [6] and virtualized environments [7]. This paper proposes to address these issues for a specific application: IVR. It defines task scheduling and computational resource sharing strategies based on genetic algorithms, in which different objectives are optimized. We chose genetic algorithms because their robustness and efficiency for the design of efficient schedulers have been largely proven in the literature [8,9]. More specifically, we identify task assignments that guarantee maximum utilization of resources while minimizing the execution time of tasks. Moreover, we propose a resource allocation strategy that minimizes substrate resource utilization and the resource allocation time. We also simulated the algorithms used by the proposed strategies and measured and analyzed their performance.

The rest of the paper is organized as follows. Section "Assumptions and problems statement" presents the assumptions and the problem statement. Section "Resource management algorithms" discusses the proposed resource management algorithms. Sections "Computational resource sharing algorithm" and "Task scheduling" describe the computational resource sharing and instantiation request

scheduling algorithms respectively. Section "Performance results and analysis" presents the main performance analysis. The state of the art review and the conclusion are given in sections "State of the art review" and "Conclusion" respectively.

Assumptions and problems statement

Assumptions

The substrate management engine (Figure 1) has two entities: the IVR resource manager and the IVR instance manager. The resource manager maintains and monitors the current state of resources and allocates resources for new IVR service substrates (ISSs). An ISS is the set of IVR substrate instances used by a single IVR application. Each ISS is managed by a separate IVR instance manager, which coordinates the process of ISS instantiation (i.e., ISS creation, configuration and activation).

We assume that each virtualization machine (i.e. a machine that hosts the substrate layer) has a fixed number of processors that are dedicated to the processing of the incoming instantiation requests, and a fixed amount of computational resources (e.g. virtual machines, processors, CPU, memory, disk space) that are used to run the different ISSs. The computational resources are shared among a set of ISSs, each having specific resource requirements evaluated in terms of CPU, memory, disk space and bandwidth. We assume that the virtualization machine capabilities are known in advance, while the ISSs' resource requirements are estimated at run time. In our case, the latter are estimated using a set of functions derived from observed measurements performed using a prototype of a virtualized infrastructure for IVR applications in the cloud [4].

When a resource manager receives an instantiation request, it verifies the availability of the substrate resources according to the ISS requirements. It then creates a new ISS and allocates the necessary resources. If no more substrate resources are available, the resource manager will reject all incoming instantiation requests.

We further assume that the service quality parameters required by each ISS are described by the IVR application provider using a service level agreement (SLA). In this work, we only consider as one SLA parameter; i.e. the satisfactory factor of the IVR application. This satisfactory factor is defined as the resources employed by a certain number of users over the allocated ISS resources. This parameter allows the control of the application status, such as ensuring that no ISS is under or over loaded and therefore guarantees pay-as-you-use access. To guarantee this quality parameter, the instantiation requests should specify the expected number of users as well as the users' arrival rate.

The IVR substrate management engine also allows the resizing of computing and network capacities, using the monitored ISS resource usage. If an ISS is over or under loaded for a certain interval of time, the resource control entity will notify the resource negotiation entity and the ISS computing and network resources are resized. The resource control and negotiation entities are out of the scope of this paper.

Problem statement

In this paper, the following notations are used:

- N is the expected number of users for a given IVR application.
- λ is the expected call arrival rate for an ISS. We should mention that λ is used in this paper only to determine the resources needed by an ISS, depending on the number of request/min that it is expected to support.
- t_n is the size in unit of time for the execution of a given instantiation request (task size). It is the time needed to instantiate, configure, and activate a new ISS.
- t_r is the time needed to compute the required resources for a given instantiation request. It is the time difference between the end of resource computation for task j and the arrival time of task j in processor queue.
- t_v is the time needed for the creation, configuration and activation of the appropriate virtual machine that will host a given ISS ($t_n = t_v + t_r$).
- m is the number of processors that can be used to handle the instantiation requests.
- (CPU_r, M_r, B_r, D_r) represents the required resources for a given ISS, in terms of CPU, memory, bandwidth and disk space, respectively.
- (CPU_c, M_c, B_c, D_c) represents the available capacities (i.e., the capabilities of the virtualization machine), in terms of CPU, memory, bandwidth and disk space, respectively.
- (τ_{cpu}, τ_M, τ_B, τ_D) represent the percentage of resource usage for a given ISS:
  - τ_{cpu} is the ratio of CPU_r over CPU_c.
  - τ_M is the ratio of M_r over M_c.
  - τ_B is the ratio of B_r over B_c.
  - τ_D is the ratio D_r over D_c.

Our objective is to propose two algorithms: computational resource sharing and task scheduling. These algorithms will be used by the IVR instance manager and the substrate IVR engine, respectively. The computational resource sharing algorithm should allow the selection of the required resources (CPU_r, M_r, B_r, D_r) for each ISS, while minimizing the amount of resources used as well as the resource allocation time, and maximizing the satisfactory factor of the ISS using a specific amount of

resources. The task scheduling algorithm should minimize the execution time for the instantiation requests (i.e., t_n), by sharing the instantiation requests among the available processors as equally as possible. No processor should be underused while others are overloaded.

Both algorithms are executed during the ISS instantiation, meaning before the IVR application is ready to receive end-users' requests. The task scheduling is first performed by the substrate IVR engine to assign the set of instantiation tasks to a given number of processors, and then each processor will run the computational resource sharing algorithm to select the resources that should be assigned to each ISS to be created. These two algorithms are described in the next section.

Resource management algorithms

Our computational resource sharing and task scheduling algorithms are based on Genetic Algorithm (GA) [10]. In GA, a population of strings randomly generated from a set of potential solutions (represented by chromosomes) is used to create new populations, based on the fitness of each individual in the population and by applying different GA operators, such as selection, crossover and mutation. The algorithm ends when a targeted fitness level is reached for the population.

In this paper, GA is used to optimize 1) the computational resource sharing, and 2) the assignment of instantiation requests to different processors provided by the virtualization machine. For each algorithm, a specific fitness function and specific GA operators are used. In the computational resource sharing algorithm, a population is represented by the resources required by each ISS to instantiate. This population is of limited size (e.g., CPU, memory, bandwidth, disk space). In the task scheduling algorithm, a population is represented by the instantiation requests. The size of the population depends on the number of instantiation requests received by the substrate provider.

We first discuss the computational resource sharing, followed by the task scheduling.

Computational resource sharing algorithm

Each processor performs resource computation for the instantiation request that is assigned to it. As a first step in the definition of the computational resource sharing algorithm for IVR applications, we performed a set of experimental measurements to quantify the resources used by a given number of ISSs. This was done using the prototype from our previous work [4]. The measurements were then used as input to define the load measurement mathematical models and the resource computation algorithm to calculate the required resources for each instantiation request since we don't have access to real arrival rate data from IVR providers. We also defined a resource computation fitness function.

Load measurement

Load measurement allows the quantification of the ISS resource usage according to the number of users accessing the ISS. It is performed to identify the required resources (CPU_r, M_r, B_r, D_r) for each ISS. The measurements were executed on a system providing a set of ISSs, and that had the following capacity: CPU_c = 1 GHz, M_c = 512 MB, D_c = 20 GB and B_c = 1 Gbps bit rate. Knowing the system capacity, we measured the used resources (CPU_r, M_r, B_r, D_r) according to different call arrival rates. The results are given in Figures 2, 3, 4 and 5. From these observed data, we derived the functional models that fit these data and that describe the relationship between the number of users and the usage of each resource CPU, BW, Memory and Disk space. We started from the models given in Equation (1) where y_{CPU}, y_M, y_B and y_D are respectively the CPU, memory, bandwidth, and disk space consumption in percentage according to the call arrival rate (here the variable λ). We propose to use a linear regression to model CPU, Memory, disk space and Bandwidth.

$$\begin{cases} y_{CPU,M,D} = a_1\lambda^5 + a_2\lambda^4 + a_3\lambda^3 + a_4\lambda^2 + a_5\lambda + a_6 \\ y_B = a_1\lambda^4 + a_2\lambda^3 + a_3\lambda^2 + a_4\lambda + a_5 \end{cases} \quad (1)$$

For each model, we computed the R-square (coefficient of determination R_s^2) to assess the accuracy of the model and how well it fits the measured data. The closer the value of R_s^2 is to 1, the better the linear regression models the data. This led to the identification of the functional parameters a_i, $i = \{1,2,3,\ldots,6\}$ presented in Equation (2).

$$\begin{cases} y_{CPU} \begin{cases} a_1 = 85\times10^{-14}, a_2 = -25.26\times10^{-10}, \\ a_3 = 28.91\times10^{-7}, a_4 = -14.48\times10^{-4}, \\ a_5 = 34.69\times10^{-2}, a_6 = 8.38 \\ R_s^2 = 0.9964 \end{cases} \\ y_M \begin{cases} a_1 = 13.30\times10^{-14}, a_2 = -361.93\times10^{-12}, \\ a_3 = 3670.03\times10^{-10}, a_4 = -17367.23\times10^{-8}, \\ a_5 = 489.26\times10^{-4}, a_6 = 12.21 \\ R_s^2 = 0.9958 \end{cases} \\ y_D \begin{cases} a_1 = -0.10\times10^{-14}, a_2 = 3.08\times10^{-12}, \\ a_3 = -19.10\times10^{-10}, a_4 = 8.14\times10^{-8}, \\ a_5 = 3.04\times10^{-4}, a_6 = 0.07 \\ R_s^2 = 0.9965 \end{cases} \\ y_B \begin{cases} a_1 = 16.92\times10^{-12}, a_2 = 73.49\times10^{-10}, \\ a_3 = -48.85\times10^{-6}, a_4 = 4.82\times10^{-2}, \\ a_5 = 11.75 \\ R_s^2 = 0.9843 \end{cases} \end{cases} \quad (2)$$

Resource computation

Resource computation allows the computation of the resources to be allocated for each ISS, in terms of CPU,

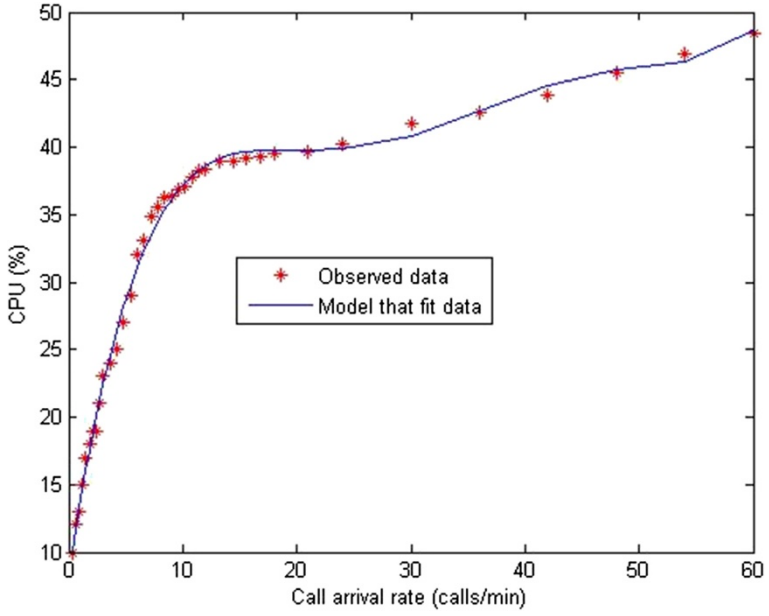

Figure 2 CPU usage.

memory, bandwidth, and disc space. It is performed using a GA-based method with the sliding window technique where two dimensional strings are used to represent the resource computation for each task in each processor.

Two dimensional strings are used to represent the resource computation for each task in each processor. One string identifies the resource combination types provided by the virtualization machine and the second identifies the required resources for each task. A resource combination type represents a possible configuration of a virtual machine that could be created by the virtualization machine. These resources can be selected separately to configure a virtual machine that will host an ISS.

In Figure 6, the resource string R_1 for instance refers to an assignment of (CPU = 1 GHz, M = 256 MB, B = 0.250

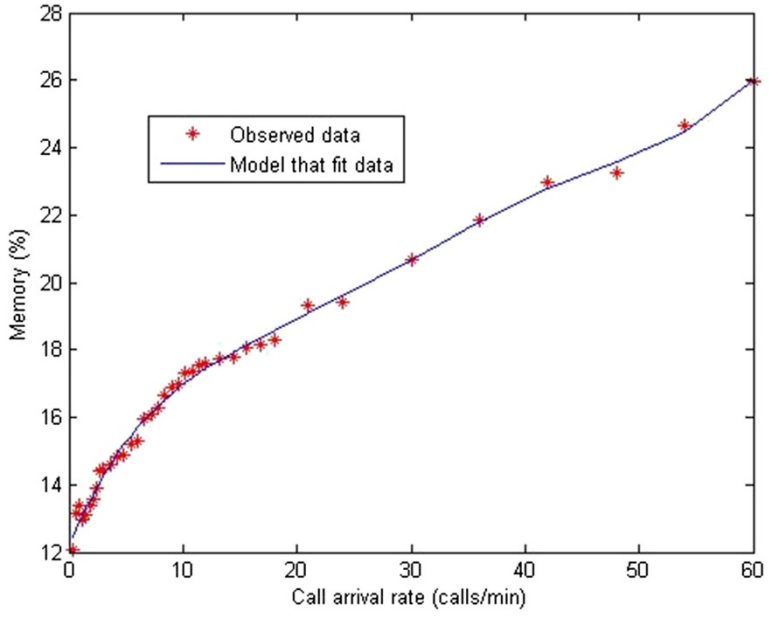

Figure 3 Memory usage.

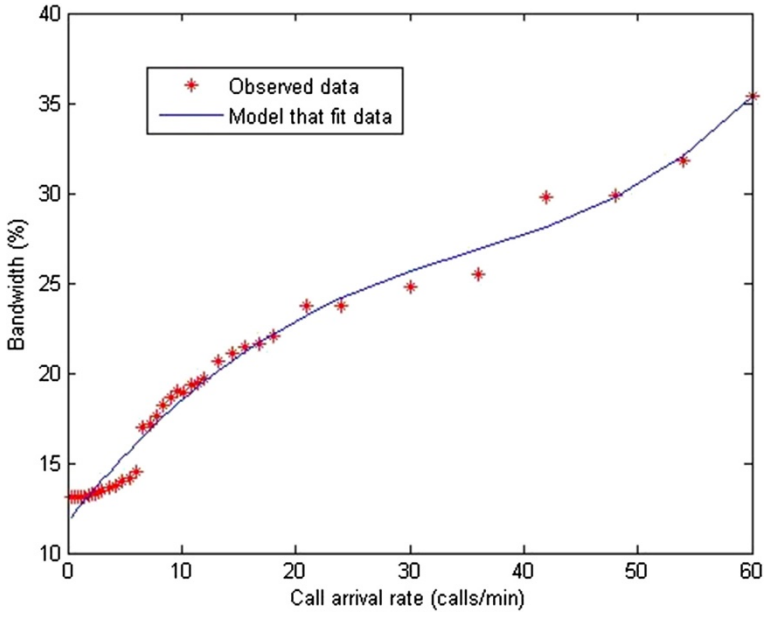

Figure 4 Bandwidth usage.

Gbps, D = 1 GB), as defined in Table 1. These are discrete values as offered by the substrate provider. A substrate provider may for instance allow to only reserving 256 MB and its multiples in terms of memory.

Figure 6 shows an example of two-dimensional strings, where each resource in the second string is identified by the resource values C_i^j , where i is the type of resource (CPU (1), memory (2), bandwidth (3) or disk space (4) and j is the resources allocated to the instantiation request. The resources to be allocated to each instantiation request are identified using the resource computation selection, crossover and mutation methods described in section 5.4.

Two main objectives are defined and used by the resource computation fitness function. The first objective is to maximize the satisfactory factor of each ISS (τ_{cpu}, τ_M, τ_B, τ_D). These values are given by Equation (3). These satisfactory factors are used as fitness functions for the GA resource computation algorithm. The closer the satisfactory factors are to 1, the better the resource usage. If the satisfactory factors are less than 0.75 or greater than 1, the

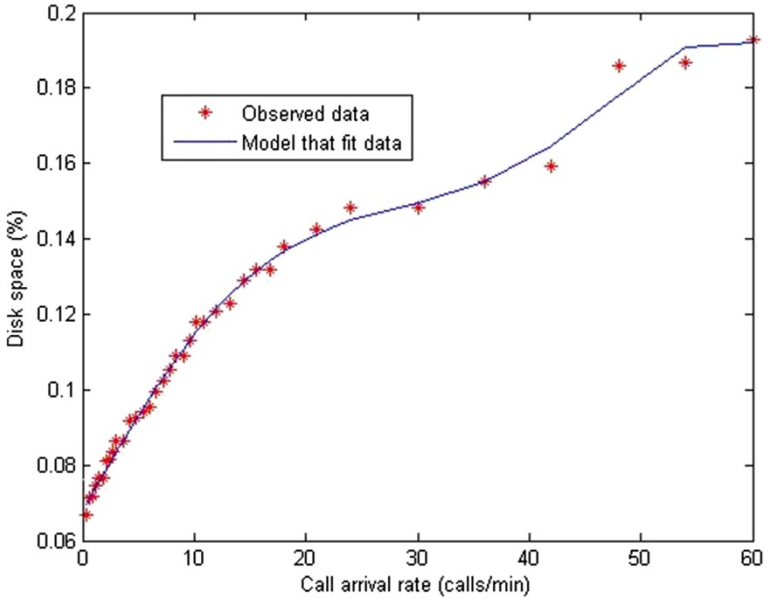

Figure 5 Disk space usage.

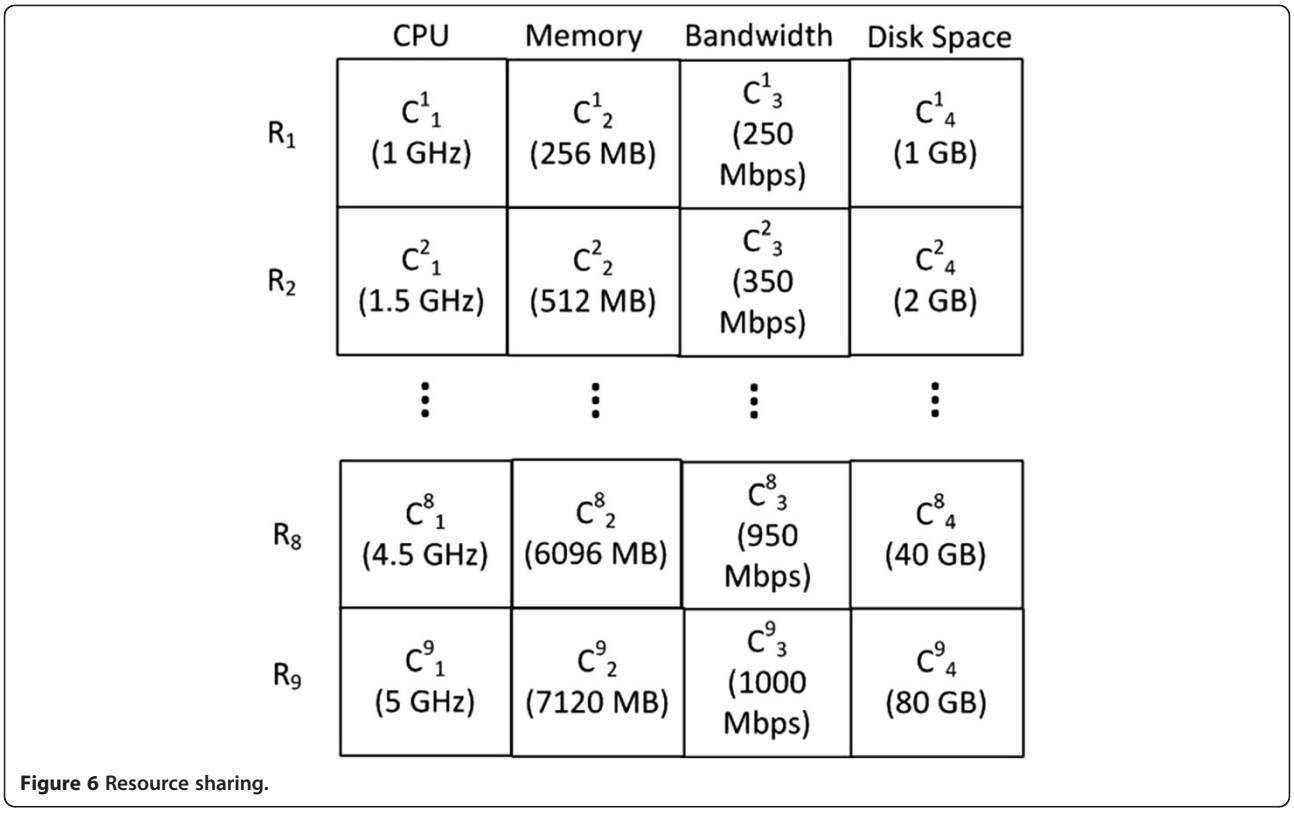

Figure 6 Resource sharing.

resource is under or over loaded, respectively. The second objective is to minimize the time t_r to compute the resources to be allocated. The resource computation selection, crossover and mutation method proposed in section 5.4 satisfies this objective.

$$\begin{aligned}\tau_{cpu} &= \frac{y_{cpu}}{C_1^j},\ j = 1, ..., m\\ \tau_M &= \frac{y_M}{C_2^j},\ j = 1, ..., m\\ \tau_B &= \frac{y_B}{C_3^j},\ j = 1, ..., m\\ \tau_D &= \frac{y_D}{C_4^j},\ j = 1, ..., m\end{aligned} \tag{3}$$

Table 1 Example of resource combination types

Type	Capacity			
	CPU (GHz)	RAM (MB)	BW (Gbps)	Disk space (GB)
1	1	256	0.250	1
2	1.5	512	0.350	2
3	2	1024	0.450	4
4	2.5	2048	0.550	8
5	3	3072	0.650	10
6	3.5	4096	0.750	20
7	4	5072	0.850	30
8	4.5	6096	0.950	40
9	5	7120	1.000	80

Resource computation crossover and mutation

The selection of strings in a population is based on the models given in Equations (1–2). Knowing the expected call arrival rate λ for each ISS, the required resources for each task are estimated using these models. For instance, if the call arrival rate for a given ISS is λ = 30 requests/min, then the expected resource usage is:

$$\begin{aligned}y_{CPU} &= 49.89\% \times 1\ GHz = 0.4989\ GHz\\ y_M &= 20.6659\% \times 512\ MB = 105.8094\ MB\\ y_B &= 25.31\% \times 1\ Gbps = 0.2531\ Gbps\\ y_D &= 0.1629\% \times 20\ GB = 0.0326\ GB\end{aligned} \tag{4}$$

The expected resource usage levels for small IVR systems are given in Table 2 (for small-size IVR system, λ ≤60 requests/min). From these expected resource usage levels we derive the resources to be allocated for each instantiation request by selecting the appropriate strings in the population described in Figure 6. For instance, an instantiation request with λ = 60 requests/min requires 1.22 GHZ of CPU, 130.74 MB of memory, 0.35 Gbps of bandwidth and 0.12 GB of disk space. For this request, the string with the closest values is selected (i.e., R_1 CPU, memory, bandwidth and disk space values). We compute the fitness values using Equation (3). We find $\tau_{CPU} = 1.222$, $\tau_M = 0.51$, $\tau_B = 1.4$ and $\tau_D = 0.12$. The values of the allocated

Table 2 Required resources according to the call arrival rate

λ requests/min	Required resources			
	y_{CPU} (GHz)	y_M (MB)	y_B (Gbps)	y_D (GB)
15	0.41	92.35	0.21	0.02
30	0.50	105.81	0.26	0.03
45	0.76	118.17	0.27	0.05
54	0.99	123.80	0.32	0.08
60	1.22	130.74	0.35	0.12

resources should be reduced for fitness values less than 0.75 and increased for a satisfactory factor greater than 1. Therefore, the CPU and bandwidth values of string R_1 (1 GHz and 0.250 Gbps) are swapped with that of string R_2 (1.5 GHz) and 0.35 Gbps) respectively. The memory and disk space of string R_1 remain unchanged because the allocated resources are the smallest values provided by the IVR substrate. The population derived from this mutation process will have a satisfactory factors $\tau_{CPU} = 0.81$ and $\tau_B = 1$. This new population will be selected to represent the resources to allocate to the received instantiation request that guarantee the best resource usage according to the resource combination types provided by the ISS substrate.

As a second example, let's consider the case of an instantiation request where $\lambda = 160$. This request requires 3.6 GHZ of CPU, 420 MB of memory, 1.05 Gbps of bandwidth and 0.3 GB of disk space. For this request, a virtual machine with resource type R6 will be selected to host the new ISS. The memory, bandwidth and disk space of string R_6 are swapped with that of strings R_2 (512 MB), R_9 (1 Gbps) and R_1 (1 GB) respectively, with satisfactory factors $\tau_{CPU} \cong 1$, $\tau_M = 0.82$, $\tau_B \cong 1$ and $\tau_D = 0.3$.

Task scheduling

The substrate IVR engine receives a set of instantiation tasks that should be assigned to a number of processors. We propose to adapt the task scheduling algorithm proposed in [6] for this purpose. Therefore, we propose to use a GA-based method to perform load balancing and a sliding window technique to initialize a population of tasks on which the GA will be applied [6]. This new algorithm is called ISI GA (Instantiation request scheduling for IVR based on GA). At each time, the tasks that are within the window are reordered using the GA selection, crossover and mutation methods described in Section Task representation selection, crossover and mutation and then assigned to the processors for execution. The window is dragged to the next group, for a repeat of the assignment process, when the tasks within the window are placed in processor queues [11]. We first introduce how the set of available processors and tasks are represented; we then describe our task scheduling fitness function, and end with the task scheduling algorithm.

Processor and task representation

The scheduling for parallel processors is represented by two-dimensional strings. One string identifies the processors and the other represents the scheduled tasks in each processor queue [11]. We identify each task with its task size $t_n = t_v + t_r$ in unit of time, preceded by the number n of tasks to be scheduled in the system (e.g. $2(t_v + t_r)$). For the example presented in Figure 7a, the processors' string will include the list of available processors. To perform the GA algorithm, the strings are converted from two-dimensional to one as shown in Figure 7b.

From the experimental measurements we carried out using the implemented prototype and 8 processors, we noticed that the t_v value is the same for all ISSs. However, the t_r differs from one ISS to another. This is one of the differences between the original scheduling and the adapted algorithms. In the original algorithm (proposed in [6] and [11]), the t_n value for each individual in a GA population is supposed to be known in advance and is the same for each individual in the population. For the ISS instantiation scheduling, the t_v is known in advance but t_r should be computed using our proposed computational resource sharing algorithm. This may result in a different t_n value for each individual in the same population.

Task representation fitness function

The fitness function allows the evaluation of the task scheduling performance according to specific objectives. The main goal here is to identify task assignments that guarantee maximum processors utilization, to balance the traffic load across processors and to guarantee minimum execution time of tasks.

In [6], the following objectives apply: 1) a minimization of the largest task completion time (i.e., Maxspan) across processors to guarantee that assignment tasks will be executed in the shortest time possible [8]; 2) increase of the average processor utilization based on the Maxspan value, and 3) optimization of the number of tasks in each processor queue, in order to ensure proper load balancing across the processors.

We propose to combine the first and the second objectives by defining the TaskSpan as the difference between the largest task completion time and the smallest task completion time among all the processors in the system. The TaskSpan is calculated as in Equation (5), where n is the number of tasks in each processor queue.

$$TaskSpan = \max_{i=1,\ldots,m}\left(\sum_{j=1}^{n}\left(t_v + t_{rj}\right)\right) - \min_{i=1,\ldots,m}\left(\sum_{j=1}^{n}\left(t_v + t_{rj}\right)\right) \quad (5)$$

In the example given in Figure 8, we assume that t_v is equal to 6 units of time and the times t_r to compute the

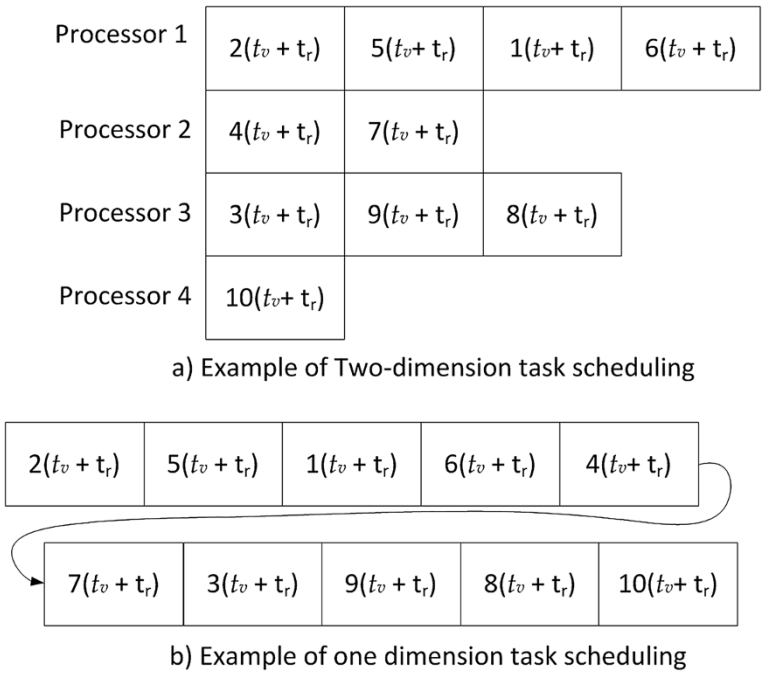

Figure 7 Task scheduling: (a) two-dimensional task scheduling strings: one identifying the processors and the other the scheduled tasks. (b) Conversion of two-dimensional task scheduling strings into one dimensional string.

required resources for tasks 1 to 10 are respectively 4, 3, 8, 9, 8, 5, 7, 10, 8 and 12.

Then, the processor 1 will execute tasks 2, 5, 1 and 6 within 44 units of time. Tasks 4 and 7 will be executed on processor 2 within 28 units of time, while tasks 3, 9, 8 and 10 will be executed within 44 and 18 units of time, respectively. Therefore, the TaskSpan for this task schedule example is 26 (Equation (6)).

$$TaskSpan = \max_{i=1,\ldots,4}(44, 28, 44, 18) - \min_{i=1,\ldots,4}(44, 28, 44, 18) = 26 \quad (6)$$

We propose to use the difference between the largest and the smallest task completion times rather than the MaxSpan value (as used in [6] and [8]) because guaranteeing a minimum TaskSpan value not only ensures a shorter task completion time but also a proper load balancing between all processors.

The second objective that we propose to define is the average processor utilization value. In [6], this value is the sum of all processor utilization levels by the total number of processors. Each processor utilization is calculated by dividing the task completion time by the MaxSpan value. The higher the average processor utilization, the better is the load balancing across the processors [6]. However, this objective does not guarantee that the load is well balanced across processors. We propose to define a

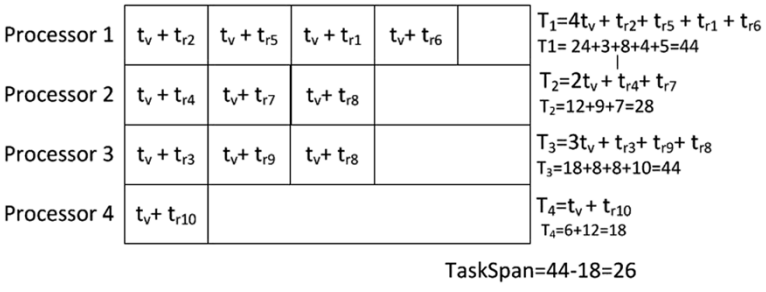

Figure 8 TaskSpan.

utilization factor U of all processors as the product of all processor utilizations.

$$U = \prod_{i=1}^{m} \left(\frac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} \right) \quad (7)$$

Using the example given in Figure 8, this will lead to:

$$\begin{cases} P_1 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{24+3+8+4+5}{44} = \dfrac{44}{44} = 1 \\ P_2 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{12+9+7}{44} = \dfrac{28}{44} = 0.64 \\ P_3 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{18+8+8+10}{44} = \dfrac{44}{44} = 1 \\ P_4 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{6+12}{44} = \dfrac{18}{44} = 0.41 \end{cases} \quad (8)$$

Therefore, the U for this task schedule will be 0.26 and the average utilization value as defined in [6] will be 0.76. If we assign task 6 to processor 4 rather than to processor 1 in order to better balance the load in term of task completion time, this lead to:

$$\begin{cases} P_1 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{18+3+8+4}{44} = \dfrac{33}{44} = 0.75 \\ P_2 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{12+9+7}{44} = \dfrac{28}{44} = 0.64 \\ P_3 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{18+8+8+10}{44} = \dfrac{44}{44} = 1 \\ P_4 : \dfrac{\sum_{j=1}^{n} (t_v + t_{rj})}{MaxSpan} = \dfrac{12+12+5}{44} = \dfrac{29}{44} = 0.66 \end{cases} \quad (9)$$

The utilization factor is then equal to 0.31, but the average utilization value as defined in [6] will remain unchanged 0.76. Therefore, the greater the utilization factor, the better the load balancing.

TaskSpan and utilization factor U are the two main objectives used by the fitness function of the GA task scheduling algorithm we propose. This function is defined as follow:

$$f = \frac{U}{TaskSpan} \quad (10)$$

The higher the fitness function, the better is the task scheduling. The single objective function f derived from this multi-criterion optimization problem reduces the problem's complexity while satisfying the multiple objectives predefined. Using a single objective function also helps in meeting our requirement on minimizing the execution time for the instantiation requests, as multi-objective functions are known to require a longer processing time.

Task representation selection, crossover and mutation

We propose to reuse the selection, the crossover and the mutation methods described in [6]. The selection operator is based on the roulette wheel method [10]. In this method, the selection of strings in a population is based on their fitness values. These values are used to assign a probability of being selected to each string. These probabilities are computed by dividing the fitness of each string by the sum of the fitness values of the current set of strings in the population. The slots of the roulette wheel are created by adding the probability of the current string to the probability of the previous string. The probabilities are then assigned until the value of 1 is reached. Then, the strings are selected randomly by generating a random number between 0 and 1. To perform the crossover operation, the selected strings are then converted from two dimensions to one. We use this two dimensions string to balance the number of tasks across the processors. For instance, for 12 tasks and 8 processors, this procedure ensures that each processor will have at least one task, and no more than 2 tasks. Hence, 8 tasks are allocated to 8 processors and 4 tasks are randomly assigned to 4 processors. This will allow the GA to converge for a fixed number of generation cycles.

The crossover operator is based on the cycle crossover method [6]. In this case, two one dimension strings S_1 and S_2 are selected. The crossover operation begins by selecting a random starting point between *1* and the length of the strings S_1 and S_2. Let us assume that this starting point is $S_{1,n}$ which denotes the task at the position *n* in string *1*. This task is marked as finished, and its corresponding task at $S_{2,n}$ is then also marked off as finished. The task in S_1whose position is the value of $S_{2,n}$ is marked as finished and its corresponding task in $S_2$is then marked off as finished as well. This process ends when the starting point $S_{1,n}$ is reached once again. Then, the remaining tasks $S_{1,n}$ that were not marked off are swapped with their corresponding tasks in S_2 (e.g., $S_{1,4}$ is

swapped with $S_{2,4}$). When all tasks in the two strings are crossed over, they are reordered and converted to a two-dimensional form to compute their new fitness values.

The third GA operator is based on swap mutation. It randomly selects and then swaps two tasks. Each task is taken on randomly selected processors which should be different in order to ensure that the two selected tasks are not the same. New fitness values are then computed using the population derived from this swapping mutation process.

Performance results and analysis

For the computational resource sharing algorithm, we propose to compare the required resources (CPU_r, M_r, B_r, D_r) with the allocated ones. The required resources are estimated based on Equations (1–2), whereas the allocated ones are estimated using the proposed GA computational resource sharing algorithm. The comparison will allow us to see if the example of resource combination types given in Table 2 and which is usually used in cloud computing environment is suitable to well manage the available resources (CPU_c, M_c, B_c, D_c).

For the instantiation request scheduling algorithm, we compute the total completion times and average processors utilization in order to compare the resource usage efficiency of the proposed fitness function with those analyzed in [6] (dynamic and random algorithms). We choose to compare with the dynamic algorithm because it is the basis of our algorithm which we enhanced; and with the random algorithm because the dynamic one was compared to it. The different proposed algorithms were simulated using Matlab and the results were measured via the same simulator.

Computational resource sharing

The required and allocated CPU, memory and bandwidth were computed according to the call arrival rate. As described in Table 2, the required resources for call arrival rates less than 60 requests/min are less than 1 GHZ of CPU, 256 MB of memory, 350 Mbps of bandwidth and 1GB of disk space. Because the performance measurements are too small for a small-sized IVR, we computed the required and allocated resources for a large-sized IVR (i.e., for $\lambda \geq 60$ requests/min).

For instance, for $\lambda = 60$ requests/min, the required CPU was almost 1.2 GHz and the estimated resource was 1.5 GHZ which represents a typical resource combination type provided by the virtualization machine. The required memory was almost 131 MB and the estimated resource was 256 MB. The required bandwidth was almost 350 Mbps and the estimated resource was 350 Mbps. For its part the required disk space was very small even for higher call arrival rates (e.g., $\lambda = 600$ requests/min) always staying under 1GB. This is due to the nature of the IVR applications, which need little disk space. The disk space measurements were therefore not included in this section because the estimated value was the same (i.e. 1 GB).

As shown in Figures 9, 10 and 11, the required and allocated resources increased as the call arrival rates were increased. These performances were expected because each IVR call requires specific ISS resources (CPU, memory and bandwidth) to be executed. In Figure 9, the difference between the required CPU and the allocated value was small and according to the CPU satisfactory factor (Figure 12) the resource usage percentage was greater than 90%.

For λ greater than 350 requests/min, this percentage was more than 95%. The required and the allocated

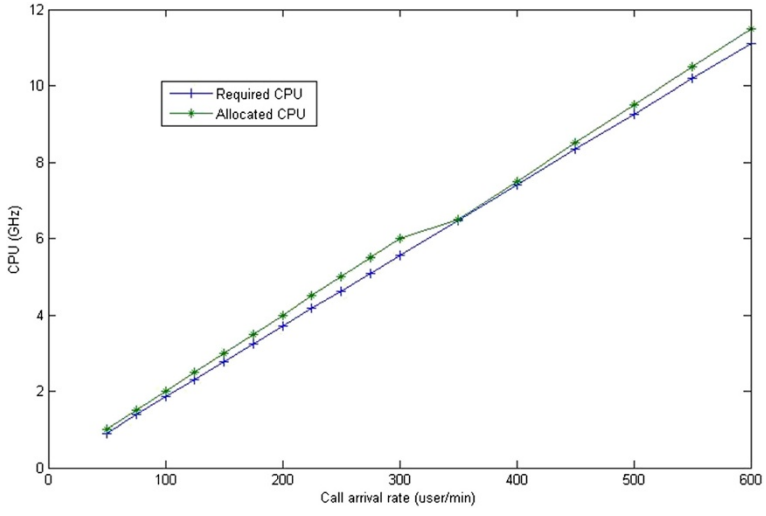

Figure 9 Required and allocated CPU.

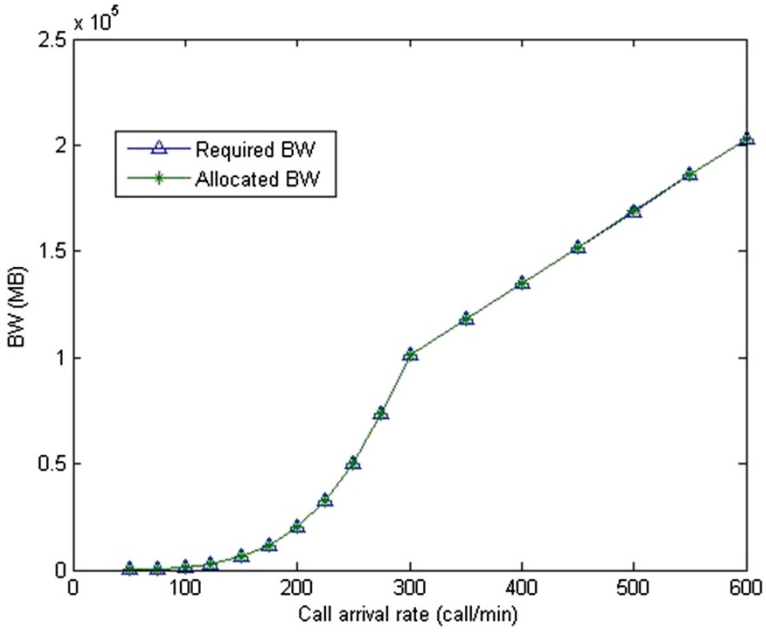

Figure 10 Required and allocated bandwidth.

bandwidth were almost the same and the bandwidth usage was almost 100% (Figure 12). The required and allocated memory measurements were different due to the fact that the sizes of the resource combination types were predefined 256, 512, 1024, etc. For instance, for $\lambda = 60$ requests/min the required memory was almost 131 MB while the allocated memory was 256. Only half the capacity memory was therefore used. For $\lambda = 100$ requests/min, the required memory was almost 500 MB and the allocated memory was 512 MB, representing 97% memory usage. Therefore, for a large-sized IVR, the higher the instantiation request arrival rate, the higher the percentage of CPU and Bandwidth resource usage. Unlike the CPU and bandwidth performance improvement in terms of resource usage, Figures 11 and 12 show that the memory usage was efficient.

In fact, the memory resource needs for an IVR application are small. The difference between the required

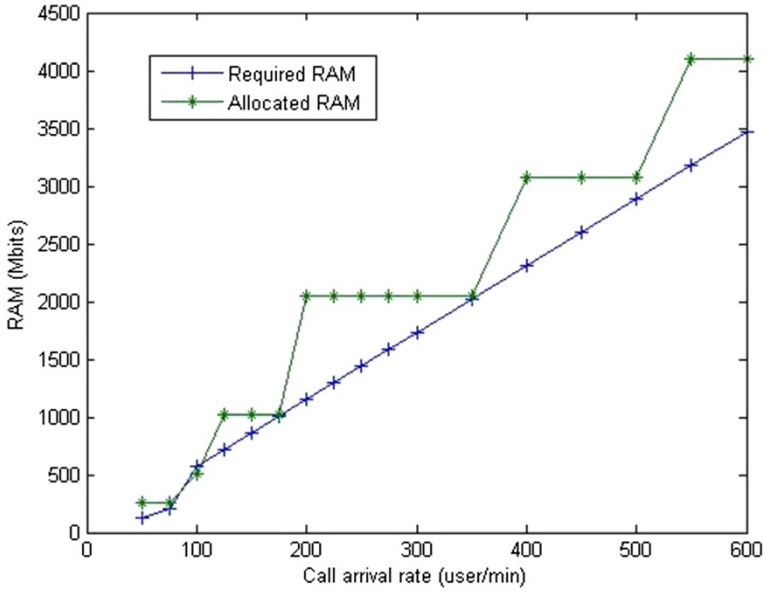

Figure 11 Required and allocated memory.

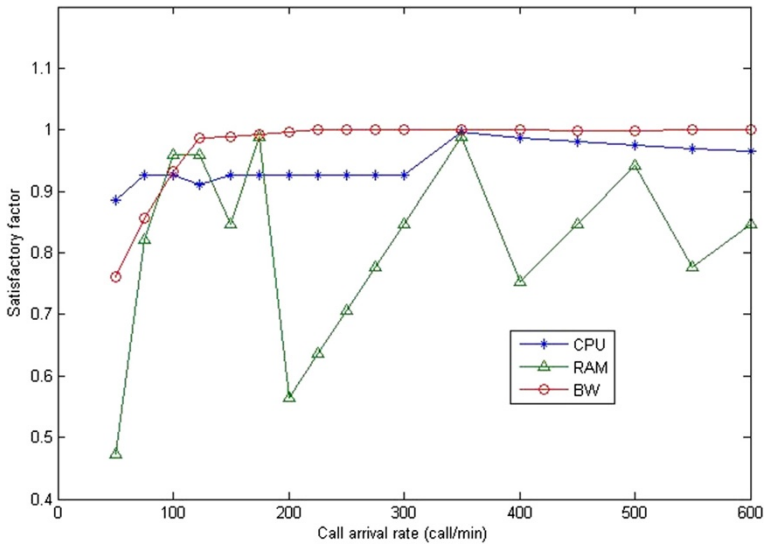

Figure 12 CPU, memory and bandwidth satisfactory factors.

memory and the allocated value varies according to the instantiation request arrival rate (Figure 11). The percentage of the resource usage varies from 50 to almost 100% for instantiation request arrival rates less than 250 requests/min and varies from 70% to 100% otherwise. We also compute the time required to allocate resources and to serve all the IVR instantiation requests. The computational resource sharing was fast because the slicing window technique allows reducing the total number of iteration of the GA algorithm. Table 3 gives some values of the required time according to the number of IVR instantiation requests.

As a conclusion, the resource combination types usually used in cloud environments do not allow for efficient resource usage. New combination types are therefore needed.

Instantiation request scheduling

We computed the total completion times and average processor utilization according to the number of tasks, the sizes of window and number of generation cycles. These performances were compared to dynamic GA as well as to random allocation strategies that were analyzed in [6].The dynamic algorithm is based on the selection, crossover and mutation methods described in 5.3.

A set of tests were performed using the following default values: 100 instantiation requests, 8 processors, window size of 10 requests, generation cycles of 10 and population size of 10. As instantiation requests' length (t_n) we used 20 units of time for the following three reasons. First, this is the average time measured using the implemented IVR prototype: t_v = 6 ms and t_r = 14 ms in average given the prototype setup. The second reason is that we wanted to compare our algorithm to those of reference [6] which use the same task size for all of the individuals in the population. Third, using a fixed t_n value will not affect the performance of our algorithm. We have also tested the proposed algorithm for different numbers of generation cycles (10, 20, and 30) and we noticed very slight changes.

To compute the total completion time and the average processor utilization, the test parameters were set to the default values and we varied the number of tasks from 0 to 1500. These values are summarized in Table 4.

As shown in Figure 13, the total completion time for the three algorithms increased as the number of tasks increased.

Hence, the higher the number of tasks to be scheduled, the longer is the total completion time. Moreover, the instantiation request scheduling algorithm provided

Table 3 Time required allocating resources

Number of IVR instantiation requests	Required time (seconds)
14	0.38
18	0.46
22	0.62
24	0.86

Table 4 Default simulation values

	Variation of number of tasks	Variation of window size	Variation of generation size
Number of processors	8	8	8
Window size	10	10 to 60	10
Generation size	10	10	10 to 60
Number of tasks	0 to 1500	0 to 1500	0 to 1500

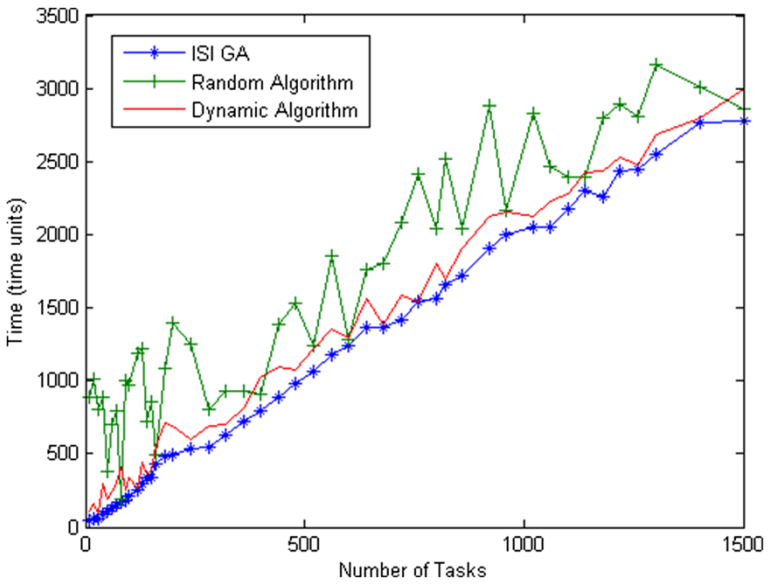

Figure 13 Completion time according to number of tasks.

a better performance than the two other algorithms. In Figure 14, the average processors utilization is much higher for instantiation request scheduling algorithm than for the dynamic and the random algorithms. The means of these utilizations were 0.83, 0.74 and 0.58 for instantiation request Scheduling, dynamic and random algorithms, respectively. These performance behaviors are due to the fact that the instantiation request scheduling algorithm provides a fitness function that guarantees a better processors utilization and the faster task execution times than those defined for the dynamic and random algorithms. It therefore requires less processing.

We also computed the total completion time and the average processor utilization according to the window size. Figures 15 and 16 both illustrate these performances for 10 tasks.

The total completion time decreased as the window size was increased and the average processor utilization improved as the window size increased for instantiation request and dynamic algorithms. Moreover, the instantiation request scheduling algorithm outperformed the dynamic

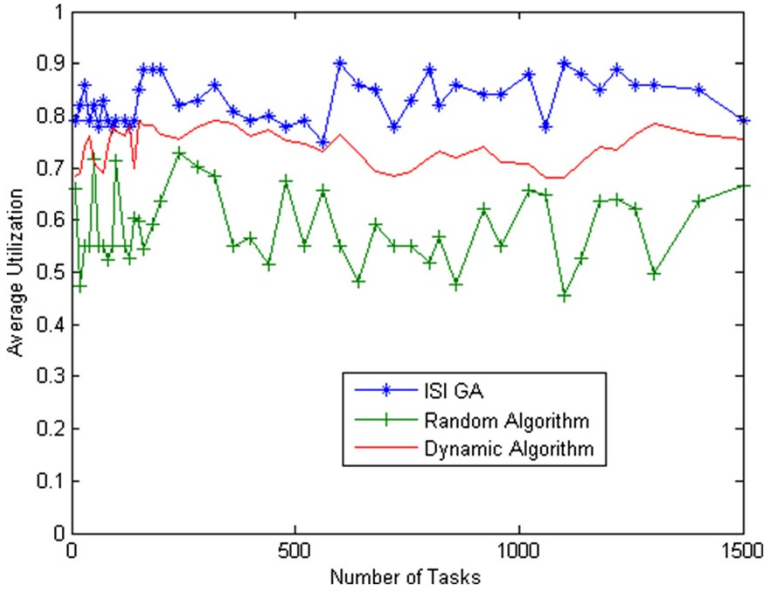

Figure 14 Processors utilization according to number of tasks.

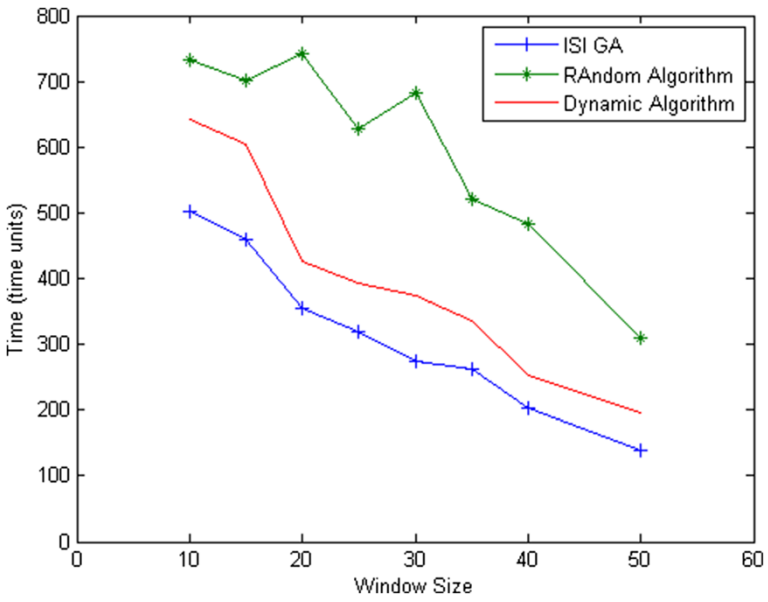

Figure 15 Completion time according to window size.

scheduling and random algorithms. This performance improvement shows that the increasing number of tasks to be scheduled was well handled by the 8 processors.

Furthermore, we analyzed the effect of the number of generation cycles on the instantiation request scheduling algorithm. We varied the number of generation from 0 to 60 for a 10 request window size and a task number of 10. Figures 17 and 18 show the total completion time and average processor utilization according to the number of generation cycles. The total completion time and the average processor utilization decrease as the number of generation cycles increase. A significant reduction in completion time and improvement in processor utilization were noticed when varying the number of generation cycles from 10 to 30. These performances were expected because increasing the number of generation cycles improves the task assignment quality.

However, through the simulation we noticed that after a certain number of generation cycles (~25 cycles), the average processor utilization results are slightly different. This

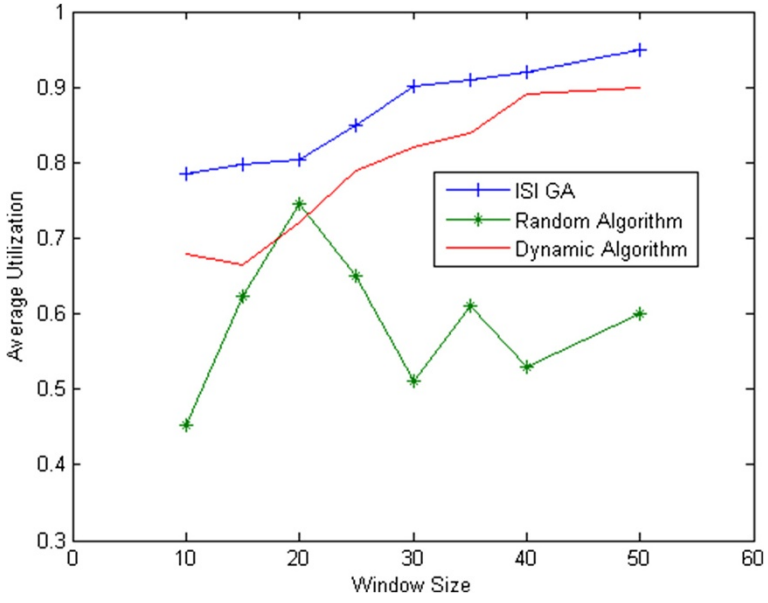

Figure 16 Processors utilization according to window size.

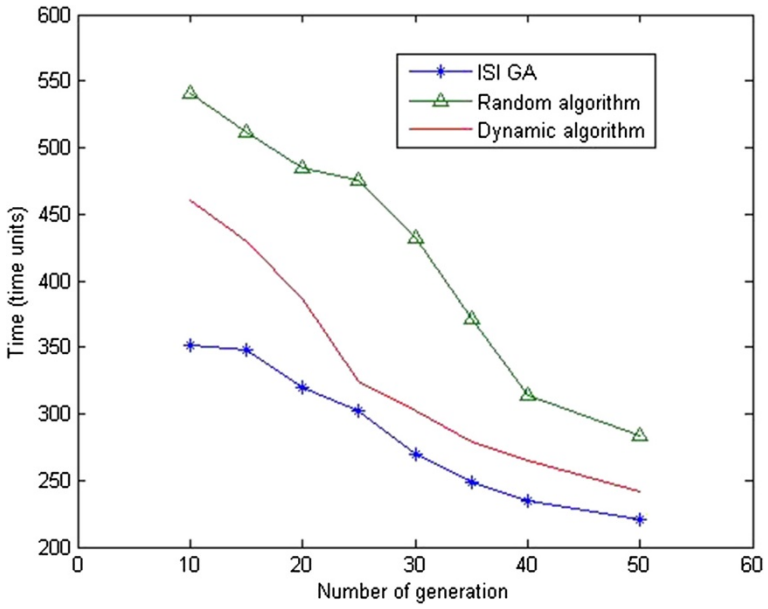

Figure 17 Completion time according to number of generation.

finding can be used to configure an upper limit for the generation cycles, in order to meet our requirement on minimizing the algorithm execution time. Moreover, the instantiation request scheduling performed better than the dynamic and random algorithms. As a conclusion, the proposed instantiation request scheduling algorithm outperforms the dynamic and random algorithms in almost all of the taken measurements. Furthermore, the algorithm performances are enhanced when the windows size is increased and the number of generation cycles increases.

State of the art review

This work joins many efforts devoted to task scheduling and load balancing across processors or computers in non-virtualized and virtualized environments. It complements them by defining new task scheduling and computational resource sharing strategies based on genetic algorithms for virtualized IVR application. Moreover, it proposes new task assignment that guarantees maximum utilization of resources while minimizing the execution time of tasks for virtualized IVR applications. We also propose a

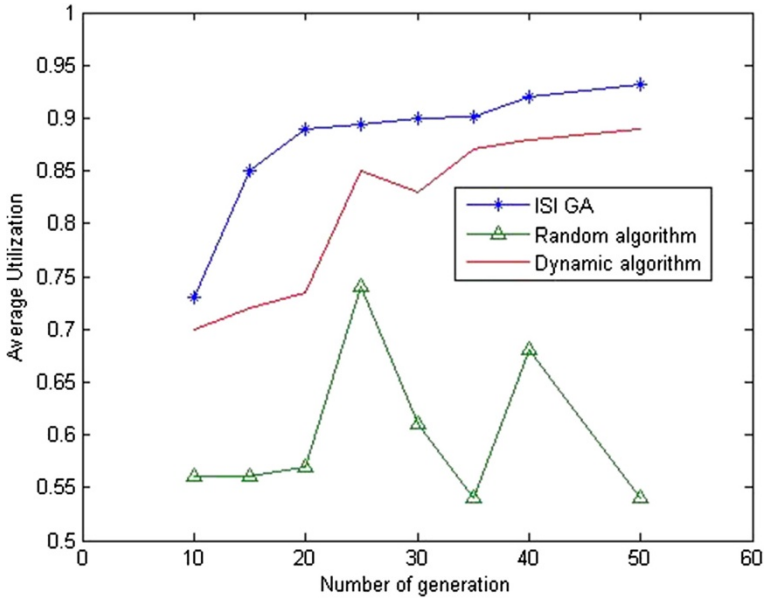

Figure 18 Average Utilization according to number of generation.

resource allocation strategy that minimizes substrate resource utilization and the resource allocation time.

The issue of load balancing based on genetic algorithm in non-virtualized environments has been addressed in [6-8]. It has been addressed by Y. Albert Zomaya et al. in [6] to propose a framework for using genetic algorithm to solve scheduling problem for parallel processor systems and to highlight the condition under which this algorithm outperform the ones based on heuristics. The genetic algorithms proposed in this paper are based on this framework.

In [11], authors propose task scheduling algorithm to achieve minimum execution time, maximum processor utilization and optimal load balancing across different processors by defining tree objective functions. In this work, we have demonstrated how two of these objective functions did not guarantee minimum execution time and maximum processor utilization and this independently of the virtualized application to which the scheduled tasks belong. We have proposed new objective functions that maximize resource utilization while minimizing task execution time. Several task scheduling methods based on modified genetic algorithm have been proposed [12-14]. In [12,13], authors propose to modify GA to control the task duplication and reduce the length of the processor queues. In [14], a modified genetic algorithm is proposed to handle task scheduling in parallel multiprocessor systems. Unfortunately, these modified algorithms yield to task scheduling time greater than that obtained with non-modified GA.

In [8], genetic-based algorithm has been proposed where a dedicated processor has been used to schedule tasks across processors. This paper has shown that this algorithm outperforms a genetic algorithm based on first-in first-out scheduling approach. However, these performances depend on the number and the distribution of tasks being executed. Moreover, it uses the same objective function as defined in [11]. We believe that this objective function that allows minimization of the largest task completion time does not guarantee the minimum execution time.

The load balancing issue based on genetic algorithm has also been addressed in virtualized computing environments [11,15,16]. In [11] and [15], authors propose to minimize task execution time by using an objective function that minimizes the largest task completion time as defined in [11] and [8].

In [16], author proposes task scheduling algorithm for Hadoop MapReduce framework. This framework is used to satisfy the data-processing needs in environments where high parallel computing and huge data storage are needed. The proposed genetic algorithm is based on statistical prediction model KCCA (Kenel Canonical Correlation Analysis) to identify the expect task execution time [16,17]. However, this paper neither describes how KCCA is used to predict the task execution time nor gives performance analysis of such algorithm. In [18], authors have described a non-genetic scheduling algorithm based KCCA and demonstrated that KCCA could be a good prediction mechanism. They stress the need for the task scheduling in Hadoop but not for optimizing resource usage in cloud environment (ex., CPU, memory, etc.). In our work, we have addressed this issue too. This issue has been also neglected by some research projects on grid computing [19,20]. Several research projects tackle the task scheduling issue in cloud for many applications like workflow and e-learning applications [21-24], but no resource optimization mechanism is provided in order to guarantee both efficient task scheduling and resource usage. Finally to the best of our knowledge, these two issues have been recently addressed in [25].

Our work is similar to this effort in that it considers the optimization of each required resource (CPU, memory, disk space, Bandwidth) according to specific applications needs: IVR application.

Conclusion

This paper proposes two resource management-related algorithms for virtualized IVR applications. The first algorithm concerns computational resource sharing, whereas the second relates to the scheduling of IVR application instantiation requests. Both algorithms are GA-based and they both consider several objectives regarding the optimization of resource usage and sharing at the substrate layer. The scheduling algorithm maximizes resources utilization while minimizing task execution time. The computational resource sharing algorithm minimizes the substrate resource utilization and the resource allocation time while maximizing the satisfactory factor of IVR applications.

The performance measurements conducted showed that the proposed algorithms are promising. Indeed, compared to dynamic and random algorithms, the proposed instantiation request task scheduling GA outperformed in terms of total completion time and average processors utilization. The computational resource sharing algorithm allows efficient CPU and bandwidth usage. However, due to the resource combination types used and because the memory resource needs are small for IVR applications, the memory resource usage was not that efficient.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

NK developed the proposed algorithms for computational resource sharing and task scheduling of virtualized IVR application, analyzed the obtained results and helped in the design of a virtualized IVR platform and wrote and

reviewed this manuscript. MS implemented and tested the proposed algorithms and helped in the development of a virtualized IVR platform. FB designed a virtualized IVR platform and helped in the development of this platform and in reviewing this manuscript. CA helped in the development of a virtualized IVR platform and in the implementation and the testing of the proposed algorithms. RG helped in the design of virtualized IVR platform and reviewing this manuscript. All authors approved the final manuscript.

Acknowledgment
NK would like to thank the research funding organization NSERC which helped us by giving us financial support to conduct this research.

Author details
[1]ETS, University of Quebec, 1100, Notre-Dame street West, Montreal, Quebec H3C 1K3, Canada. [2]Concordia University, 7141, Sherbrook street West, Montreal, Quebec H4B 1R6, Canada.

Received: 22 May 2014 Accepted: 1 October 2014
Published online: 15 October 2014

References

1. Xu S, Gao W, Li Z, Zhang S, Zhao J (2010) Design of Hierarchical and Configurable IVR System. Second International Conference on Computational Intelligence and Natural Computing Proceedings (CINC), pp 205–208
2. Khan A, Zugenmaier A, Jurca D, Kellerer W (2012) Network Virtualization: A Hypervisor for the Internet? IEEE Commun Mag 50(1):136–143
3. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M (2009) A break in the clouds: towards a cloud definition. ACM SIGCOMM Comp Commun Rev 39(1):50–55
4. Belqasmi F, Azar C, Soualhia M, Kara N, Glitho R (2011) A Virtualized Infrastructure for Interactive Voice Response Applications in the Cloud. ITU-T Kaleidoscope the Fully Networked Human - Innovations for Future Networks and Services, pp 1–7
5. Belqasmi F, Azar C, Soualhia M, Kara N, Glitho R (2013) A case study of Virtualized Infrastructure and its accompanying platform for IVR Applications in Clouds. IEEE Network Mag 28(1):33–41
6. Zomaya YA, The YH (2001) Observation on using genetic algorithms for dynamic load-balancing. IEEE Trans Parallel Distributed Syst 12(9):899–911
7. Xhafa F, Carretero J, Abraham A (2008) Genetic Algorithm Based Schedulers for Grid Computing Systems. Int J Innovative Comput, Inf Control 3(5):1–19
8. Kidwell MD, Cook DJ (1994) Genetic Algorithm for Dynamic Task scheduling. Proc. IEEE 14th Annual International Phoenix conference on Computers and communications, pp 61–67
9. Carretero J, Xhafa F (2007) Genetic algorithm based schedulers for grid computing systems. Int J Innovative Comput, Inf Control 3(6):1–19
10. Goldberg DE (1989) Genetic algorithms in search, optimization, and machine learning", Reading, Mass. Addison-Wesley, ISBN 0201157675
11. Zomaya AY, Ward C, Macey B (1999) Genetic Scheduling for parallel processor systems: Comparative studies and performance issues. IEEE Trans Parallel Distributed Syst 10(8):795–812
12. Kaur K, Chhabra A, Singh G (2010) Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system. Int J Comput Sci Secur 4(2):149–264
13. Omara FA, Arafa MM (2010) Genetic algorithms for task scheduling problem. J Parallel Distributed Comput 70(1):13–22
14. Probir R, Mejbah Ul Alam MD, Nishita D (2012) Heuristic based task scheduling in multiprocessor systems with genetic algorithm by choosing the eligible processor. Int J Distributed Parallel Syst (IJDPS) 3(4):111–121
15. Prabhu S (2011) Multi-Objective Optimization based on genetic algorithm in Grid Scheduling. Int J Advanc Res Technol 1(1):54–58
16. Tayal S (2011) Tasks scheduling optimization for the cloud computing systems. Int J Advanc Eng Sci Technol 5(2):111–115
17. Bach FR, Jordan MI (2003) Kernel independent component analysis. J Mach Learn Res 3:1–48
18. Ganapathi A, Kuno H, Daval U, Wiener J, Fox A, Jordan M, Patterson D (2009) Proceedings of IEEE International Conference on Data Engineering., pp 592–603
19. Ganapathi A, Chen Y, Fox A, Katz R, Katz R, Patterson D (2010) Statistics-driven workload modeling for the cloud. 26th IEEE International Conference on Data Engineering, pp 87–92
20. Gao Y, Rong H, Huang JZ (2005) Adaptive grid job scheduling with genetic algorithms. Elsevier J Future Generation Comp Syst 21(1):151–161
21. Kim S, Weissman JB (2004) A genetic algorithm based approach for scheduling decomposable data grid applications. International Conference on Parallel Processing, pp 406–413
22. Barrett E, Howley E, Duggan J (2011) A learning architecture for scheduling workflow applications in the cloud. 9th IEEE European Conference on Web Services, 83, pp 83–90
23. Yu J, Buyya R (2006) Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. Sci Programming J 14(3):217–230
24. Morariu O, Morariu C, Borangiu T (2012) A genetic algorithm for workload scheduling in cloud based e-learning. In Proceedings of the 2th International Worshop on Cloud Computing Platforms., pp 1–6
25. Zhong H, Tao K, Zhang X (2010) An approach to optimized Resource scheduling algorithm for Open-Source Cloud Systems. The 5th Annual China Grid Conference, pp 124–129

doi:10.1186/s13677-014-0015-3
Cite this article as: Kara *et al.*: **Genetic-based algorithms for resource management in virtualized IVR applications.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:15.