Application-Optimal Resource Allocation from Multiple Clouds

Jong Hoon Ahnn[1], Hao Shi[2], Liyin Tang[2], Jelena Mirkovic[3] and Ted Faber[3]

[1]University of California at Los Angeles, CA, USA
[2]University of Southern California, CA, USA
[3]Information Sciences institute, CA, USA

We believe that the future cloud market will feature numerous resource providers that will diversify their offers in terms of unique services and in terms of price. In this competitive market, application developers must select such a combination of resources, possibly from multiple clouds, that best meets application goals such as high throughput, high reliability and low cost. We call such allocation *application-optimal*. This paper investigates two approaches to application-optimal cloud resource allocation. The first approach, Back-Tracking (BT), performs a near-exhaustive search of all the possible resource combinations to select the one that best fits the application's goals. It is guaranteed to find the best allocation, but its running time increases exponentially with the resource variety. The second approach deploys a Genetic Algorithm (GA), to find promising resource combinations. It cannot guarantee that the optimal allocation will be found, but its running time is bounded. We customize these generic optimization approaches to the cloud resource allocation problem and evaluate them through simulation. Our results show that for small problem sizes the BT solution gives optimal results and that for larger problems GA, modified with problem-specific knowledge finds good results within bounded time.

I. Introduction

Many cloud providers such as Amazon, Microsoft and Google offer pay-as-you-go computing resources to customers. There is a large variation in the computing resources' capacity such as CPU speed, memory size and speed, storage type, size, and speed, as well as in price [11]. We propose a resource allocation framework that provides applications with a set of resources from multiple providers that meets their requirements and optimizes their performance according to their constraints. We call such allocation *application-optimal*.

Traditional resource allocation research focuses on maximizing a single QoS parameter – throughput or execution time [5][6], possibly while satisfying multiple constraints [7]. Most resource allocation approaches perform optimization from the resource provider's perspective, i.e. across multiple allocation requests. For example, a resource provider may seek to schedule jobs on his machines so to minimize the overall execution time of the entire batch. Conversely, we optimize for one given customer across multiple providers.

1) *Problem Formulation*

We assume that there is a single application with a set of application components that need to be allocated to physical resources from a diverse resource pool. This application has some potentially conflicting goals that it would like to meet through resource allocation, e.g., it would like to achieve the best throughput for the lowest price. The application also has certain constraints for its goals, e.g., it specifies the minimum throughput and maximum cost requirements. Since we are performing resource allocation from the customer's point of view we may lack reliable information about any resource's ability to meet application goals, save for the cost which is always given truthfully by the provider. For example, Li et al. showed that cloud resources differ widely in performance for a given price, across different providers [11]. We assume that any cloud resource type can be profiled periodically by the resource allocator for certain common tasks (such as Web queries, Database queries, etc.), and for certain application goals (e.g., run time, throughput, reliability). We further assume that the customer can express the needs of any application component in terms of a mix of profiled dimensions. The allocation process then aims to find the best match between the resource types and the application components based on their profiles, and while optimizing for global application goals and honoring global application constraints. The size of the search space equals the number of all possible assignments of resource types to application components. It grows exponentially with the sizes of applications and the variety of cloud resources, and the allocation problem is NP-hard.

2) *Solution Sketch*

In this paper we investigate two approaches to solving the above-specified resource allocation problem. Our first approach, the *Back-Tracking (BT) scheme*, performs a near-exhaustive search of the solution space, but guarantees that the optimal solution will be found. This scheme is well-suited for small search spaces. Our second approach, the *Genetic Algorithm (GA) scheme*, applies GA search techniques to reduce the search space. It does not guarantee optimality but bounds the execution time. This scheme is well-suited to large search spaces.

The back-tracking (BT) algorithm detailed in [1] has been heavily utilized in the constraint satisfaction problem (CSP) [8]. While the worst-case complexity of backtrack search is exponential, several techniques have been proposed in the literature to reduce its average-case complexity [16].

Genetic Algorithms (GA) [3][9] are known to provide robust search techniques that allow near-optimal solutions to be derived from a large search space in polynomial time, by applying the principle of evolution. In our Genetic Algorithm (GA) scheme we customize classical GA operations – crossover, mutation and elitism – to uniquely fit our resource allocation problem. We compare the performance and cost of the BT and GA schemes through simulation and show that GA is overall the better choice for cloud resource allocations. We also further propose a variation of GA that uses knowledge of application bottlenecks during its evolution steps, and show how this results in improved GA cost.

II. SYSTEMS MODELS AND PROBLEM FORMALIZATION

This section presents formal models of cloud computing resources, applications, workloads, and the resource allocation algorithm, which we use in our work.

1) *Cloud Provider Model*

We regard each cloud provider as having a set of physical machines, each of which can host multiple virtual machines (VMs). A VM is called a *resource* and can be allocated to one application component. All resources can be grouped into resource types based on their advertised specifications (e.g., AWS small instance [17]). We further define a set of resources R to which we map application components as:

$$R = \bigcup_{i=1,..,CP} \bigcup_{k=1,..,types(i)} R_u \mid 1 \le u \le N_k \tag{1}$$

where R corresponds to a set of virtual machines available over any cloud provider, CP is the number of cloud providers, *types(i)* is the number of different resource types at the provider i and N_k is the number of resources of type k. Figure 1 illustrates two cloud providers, each offering two types of resources with three instances of one type and two instances of the other type.

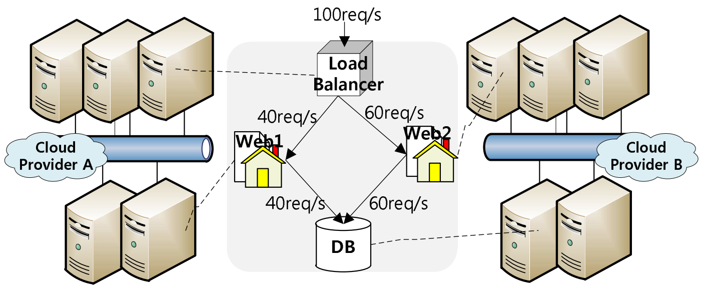


Figure 1. An simple example of Web application running on four machines; two from cloud provider A and the others from cloud provider B. A solid line is a message path between application components, and a dotted line stands for resource mapping between an application component and a machine.

2) *Application Model*

We model an individual application as a Directed Acyclic Graph (DAG) of workers (distributed components) that exchange information to perform a task. Each worker plays a specific role in the application and is matched during allocation to a resource that can achieve some estimated performance in that role, according to the application component's profile and the resource's profile. An application to be deployed in cloud computing resources is defined as $A = \{W, E\}$, where W is the set of workers and E is a set of directed weighted edges of the form (w, v, *weight(w,v)*) for *weight(w,v)* $\in$ (0,1], (w, v) $\in$ W. The weights of the edges represent the proportion of the load a given worker passes on to its children in the DAG.

3) *Workload Model*

We make two assumptions with respect to a workload conservation rule. (1) We assume that the load is exactly conserved: the weights of the outbound edges from a worker sum to one (e.g., the load balancer distributes a weight of 0.4 to web1 and 0.6 to web2 in Figure 1, which generates 100×0.4 req/s and 100×0.6 req/s respectively.), and (2) we assume that the number of requests that enter one worker equals the number of requests leaving this worker if its resources were unlimited (e.g., 40 Web requests/second entering a web server results in exactly 40 DB requests/second leaving the server and going to a DB).

4) *Problem Definition*

Resource allocation algorithm should select a combination of resources from R to assign to application workers W. Each worker in W plays some role in the application, such as being a Web server. We express this through function *role*(w) for $w \in W$. A role is approximated through a weighted combination of simple profiling tasks:

$$role = \sum_i wt_i \cdot task_i \tag{2}$$

where wt_i is the weight of the $task_i$ and all weights add up to 1.

Each resource has an estimated performance for different roles, calculated by weighted sum of its performance on each of the profiling tasks. Resources can be profiled along multiple dimensions such as throughput, reliability, input/output delay, etc, which we all subsume in term *performance*. We denote the estimated performance of resource r for role *role(w)* as *capacity*(*role*(w), r). Each resource also has a fixed cost, independent of its role. The problem we address is to map workers to resources in a way that maximizes the overall application's performance and minimizes its cost.

The function *map*(w) for $w\in W$ is generated through the allocation process, by the algorithms we propose, and outputs the selected resource for each worker. We also define the set-valued function *parents*(w) for $w \in W$ that returns the set of all $p \in W$ such that the edge (p, w, *weight*(p, w)) is in E. These are the direct parents of node w in the DAG. To simplify our formulation, without loss of generality we construct a lattice from our DAG, by creating a source and a sink node. These nodes have roles of "source" and "sink," and are assigned virtual resources of zero cost and effectively infinite capacity for only those roles.

III. FITNESS FUNCTION

In this section, we first assume that the desired optimization goal is to maximize application throughput while minimizing the cost, and we show how we construct the fitness function that evaluates the goodness of a specific resource combination. We next show how this can be generalized for any optimization goal. During resource allocation parent nodes are assigned resources either before or at the same time as their children. The throughput function is recursively defined as:

$$thr(w) = \min_{w \in W} \begin{pmatrix} capacity(role(w), map(w)), \\ \sum_{p \in parent(p)} weight(p,w) \cdot thr(p) \end{pmatrix} \tag{3}$$

That is, the throughput of a worker is the minimum of the capacity of the resource that is assigned to for that particular role, or the load imposed on it from its parents. The throughput of the application is the throughput of the sink node as:

$$thr_{system} = thr(w) \mid role(w) = \text{sink} \tag{4}$$

The cost of the application is the sum of assigned resource costs:

$$\text{cost}_{system} = \sum_{w \in W} \text{cost}\ (map\ (w)) \tag{5}$$

Our goal is to find an optimal solution that gives us the maximum throughput for the best cost. We express this optimization goal by defining a fitness function to be maximized:

$$fitness_{system} = \frac{thr_{system}}{cost_{system}} \tag{6}$$

There can also be user-defined global constraints in our formulation. We define α to be the maximum cost the user is willing to pay and β be the minimum throughput the user is willing to tolerate. At each iteration of our algorithms, we discard solutions whose throughput and cost do not honor these limits.

1) *Extension of our Approach to more Criteria*

Throughput and cost are only two of many allocation objectives that one may want to optimize. For each additional objective we will need to define a way to calculate the value for the entire application using values for individual nodes in DAG, and the resources currently allocated to them, such as is done in Eq. 4 and Eq. 5 for throughput and cost. For example, if we were to optimize reliability we would add the following equation:

$$reliability_{system} = \min_{w \in W}\left(reliablity\left(map(w)\right)\right) \tag{7}$$

We would then modify the fitness function calculation to include the sum of all objectives we want to maximize in the numerator and the sum of objectives we want to minimize in denominator. We could further introduce weights in sums to value some goals more than others.

IV. Proposed Resource Allocation Schemes

In general, the optimal resource allocation problem belongs to a class of NP hard problems. We explore two complementary approaches: back-tracking and genetic algorithms, as customized for our problem. We assume that the application developer uses Distributed Application Description Language (DADL) proposed in [10], to specify his application's optimization objectives and constraints including maximum budget, minimum throughput, and minimum reliability. Based on these input values, our framework should evaluate the search space and dynamically choose either BT or GA approach to solve the problem. At the end, the BT outputs the optimal resource assignment for cloud applications, while the GA generates a set of highest-quality solutions from its final generation.

1) *Back-Tracking Algorithm for Resource Allocation*

Our Back-Tracking (BT) approach includes two stages: topological sorting and searching/trimming of the solution space. In the first stage, a sorted worker sequence, $W_{ordered} = [w_1, w_2, ..., w_n]$, is generated from DAG based on topological sorting, which serves as one of inputs to the next searching stage.

Figure 2 gives the flowchart of the second stage: search and prune. First the variable *level* is initialized to 1 to track recursive depth, i.e. the *level*-th worker in $W_{ordered}$; and *height* represents the total number of workers, i.e., the maximum recursion depth. When assigning a resource (*r*) to one worker (second step), an optimal resource list, *oList*, will be calculated and sorted for that worker based on the fitness value, e.g. from Eq. 6.

The first resource in *oList* will be selected as the current assignment (third step in Figure 2). After that, BT checks if *level* exceeds *height*, which means BT finds a complete solution for all workers or not. If yes, this current solution, *curSol*, will replace the best solution, *bestSol*, supposing it has a better fitness value. If the whole solution space has not been thoroughly examined yet, BT backtracks to the most recent worker which still has candidates not considered in *oList*. The next candidate will go through a pruning function *TrimBranch(r)* which guarantees the current solution subjects to global constraints (performance, cost, reliability, delay, etc.) if substituting the candidate for the current worker assignment. Otherwise, the candidate will be trimmed off and the next but one resource is considered.

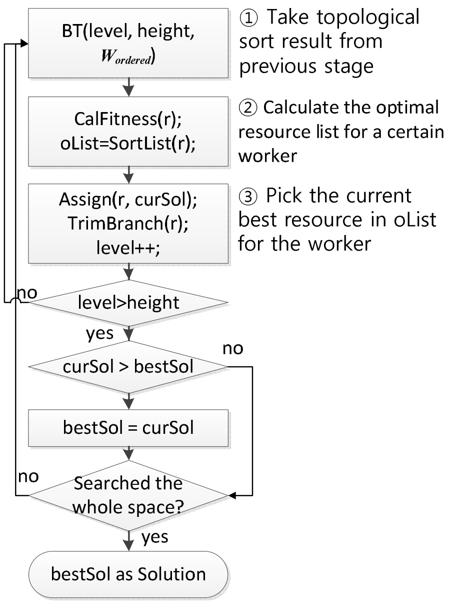

BT(level, height, W_ordered)
① Take topological sort result from previous stage
CalFitness(r); oList=SortList(r);
② Calculate the optimal resource list for a certain worker
Assign(r, curSol); TrimBranch(r); level++;
③ Pick the current best resource in oList for the worker
level>height
no
yes
curSol > bestSol
no
bestSol = curSol
Searched the whole space?
no
yes
bestSol as Solution

Figure 2. Block diagram of the Back-Tracking scheme.

We improved BT's efficiency provided that constraints are performance and costs only. In case the capacity of currently assigned resource for a worker surpasses the summation of all incoming requests from its parents, the remaining candidates in *oList* can be discarded, since they cannot improve throughput but merely increase costs. After examining or trimming all possible allocations, *bestSol* stores the ultimate optimal solution and BT terminates.

2) *Genetic Algorithm for Resource Allocation*

Our genetic algorithm (GA) approach maintains a population of chromosomes (possible resource allocations) that evolves over generations. The quality of a chromosome in the population is determined by the fitness function in Eq. 6.

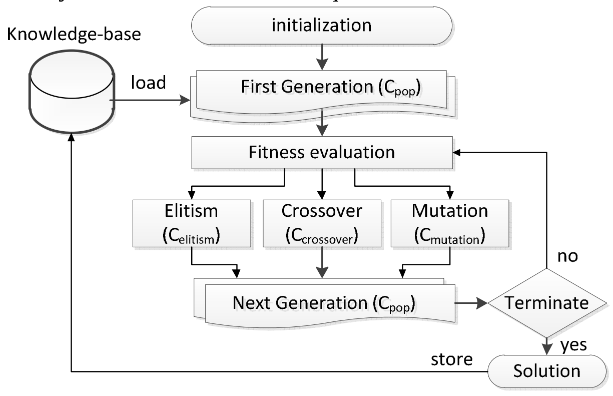


Figure 3. Block Diagram of the Genetic Algorithm scheme.

We define a chromosome as a sequence of resources given by the function *map*(*w*), which maps each worker *w* from the application DAG to some resource *r*.

We define C_{pop} as the initial population of chromosomes, C_{elite} as a set of elite chromosomes with highest fitness values that will be copied intact into the next generation, C_{cross} as a set of chromosomes that are generated performing a crossover (combination) operation on chromosome pairs from the current generation, and C_{mu} as a set of chromosomes that are generated by performing mutation operation on chromosomes from the current generation. Figure 3 presents the generic GA block diagram.

*Initialization***.** The algorithm generates the first generation C_{pop} of chromosomes, where each gene (worker to resource mapping) is randomly generated from the pool of available resources and the application's DAG.

Fitness Evaluation. This step evaluates fitness values for each chromosome in the current generation using Eq. 6.

Elitism. This step finds the $|C_{elite}|$ chromosomes with the highest fitness values and promotes them into the next generation.

*Crossover***.** This step selects two chromosomes from the elite population C_{elite} to generate the new generation C_{cross}. The combination of parent chromosomes to generate a child is done by randomly selecting a point in the sequence and combining the portion of one parent up to that point with the portion of another parent following that point. It repeats until it produces a sufficient number of candidate chromosomes.

*Mutation***.** This step randomly selects a chromosome from the current population C_{pop}, and then randomly selects the genes (mappings) to be changed. Each gene may be selected with the probability p_{mu}. To change a gene we select a new resource for the given worker from a set of currently available resources. This step ends when $|C_{mu}|$ new chromosomes have been generated.

*Evolution***.** Let C_{pop} be the current population $C_{elite} \cup C_{cross} \cup C_{mu}$. Fitness evaluation, elitism, crossover and mutation steps are repeated until we reach the pre-set number of generations at which point some small number of solutions with the highest fitness value are displayed to the user.

3) Knowledge-based GA for Resource Allocation

Kanoh et al. used the knowledge-based GA to solve university timetabling problem by taking advantage of prior years' timetables [13]. Ko et al. proposed the GA technique with the knowledge base to optimize association words [14]. As a practical work, [15] applied the same technique in mobile robots for path planning.

When investigating the performance of our GA approach we discovered that assignment of resources to throughput-bottleneck nodes in DAG dictates the goodness of a chromosome. If the resource assigned to the bottleneck was chosen so that it has sufficient capacity the overall chromosome had a high fitness value. We leverage this insight in knowledge-based GA where we first discover bottleneck nodes (for throughput) and then assign them those resources that were found to perform well in that role in the past. This speeds our algorithm because it investigates only those solutions with high fitness values.

In Figure 3, the knowledge-base is defined as a set of candidate partial solutions of the final solution. The knowledge of our interest is bottlenecks in throughput for any application. Intuitively, bottleneck nodes should be assigned the most powerful resources to minimize the limit placed on the application's performance. We characterize workers by adopting Kleinberg's hub-authority model [12].

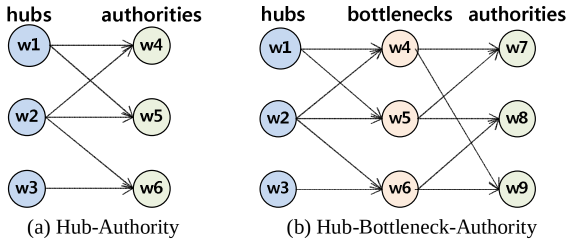


Figure 4 A densely linked set of hubs, bottlenecks and authorities. An object with unique id refers to an individual worker (w1 to w9) in an application scenario.

In Figure 4(a), the idea behind hubs and authorities is that a good hub (e.g., w1, w2 and w3) is a worker that leads to many other workers, and a good authority (e.g., p4, p5 and p6) is a worker that was linked to by many different hubs. The "good" authority is determined by evaluating the score defined in Eq. 8. The same rule holds in the "good" hub evaluation. In reality, one may pick the top *n* entities as "good" candidates. We extend this idea by adding a concept of *bottlenecks* in Figure 4(b). We further define *bottlenecks* to be workers identified by both *hubs* and *authorities*. The bottleneck workers must process lots of requests coming from hubs and pass them to authorities.

Kleinberg also proposed the HITS algorithm which iteratively calculates authority score and hub score in [12]:

$$authority(w) = \sum_{v \in V} hub(v) \qquad hub(w) = \sum_{u \in U} authority(u) \tag{8}$$

where the authority score for a worker w is equal to the sum of the hub scores of each worker $v \in V$ that points to it. The hub score for a worker w is equal to the sum of the authority scores of each worker $u \in U$ that it points to. We can augment this idea to use in bottleneck worker identification. Informally, a worker *w* is a *bottleneck* worker $w \in V \cap U$ (e.g., w4, w5 and w6 in Figure 4(b)) if *w* is a *hub* and an *authority*. The degree that the worker *w* serves as a *bottleneck* is quantified by

$$bottleneck(w) = (authoriity(w), hub(w)) \tag{9}$$

We further define *similarity* to compare one bottleneck from the application's workflow with another from the knowledge-base as

$$similarity(w_1, w_2) = \frac{\sqrt{(authority(w_1) - authority(w_2))^2} + \sqrt{(hub(w_1) - hub(w_2))^2}}{2} \tag{10}$$

The perfect match between two bottlenecks: w_1 and w_2 is given zero. The knowledge-based GA approach searches the perfect match for bottleneck workers from the knowledge-base, and then pre-assigns them to those resources that perform best in that worker's role.

V. Evaluation Setup

1) Random Resource Generator

To simulate our resource allocation framework, we randomly generate 305 resource profiles, each with unlimited quantities, along four dimensions (Web-, DB-, CPU- and Storage-intensive tasks) as illustrated by Algorithm 1. The function *Rnd* takes three parameters; min, max, and scale, and returns *random*(min, max)×scale.

Algorithm 1 Random Resource Generation algorithm

```
1: for (cap=100; cap≤400; cap++;)                 // capacity range
2:   for (c=1; c≤max; c++;)                       // dummy variable
3:     capWeb = cap;                              // capacity for Web worker
4:     capDb = cap×Rnd(5, 15, 0.1);               // capacity for DB worker
5:     capCpu = cap×Rnd(5, 15, 0.1);              // capacity for CPU worker
6:     capStorage = cap×Rnd(5, 15, 0.1);          // capacity for storage worker
7:     cost = cap×Rnd(8, 12, 0.1)/100;            // monetary cost
```

2) Four Application Scenarios

We apply our algorithms to four types of frequently-used applications [4] in cloud environments: Web application (Web), e-commerce application (eCom), a neuro-science (fMRI) application, and a map-reduce (MapR) application as shown in Figure 5 with weights of the children nodes being equal to 1/(Number of children). The algorithm maximizes $fitness_{system}$ in Eq. 6, while satisfying the global budget constraint α and the local performance constraint β specified in Table 1. The overall cost constraint α is specified as the product of the average cost over 305 profiles and the number of total workers.

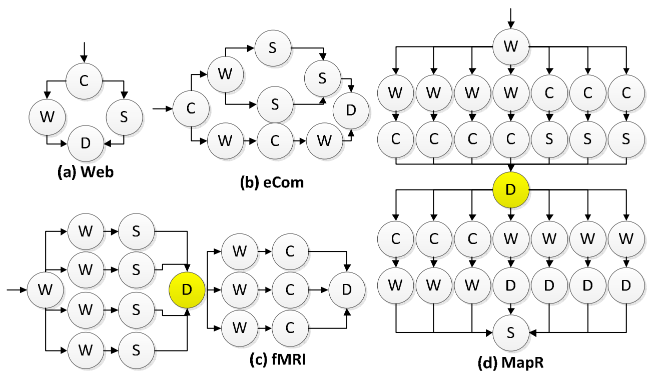


Figure 5. DAGs of the four application scenarios. Four capital letter: W, D, C, and S denote Web server, DB server, CPU-oriented worker, and storage server, respectively.

Table 1. Optimization Constraints for four application scenarios.

Constraint	Web	eCom	fMRI	MapR
α ($)	11	26	46	89
β (req per s)	150	150	150	150

3) *GA Parameters*

For the GA approaches we evaluate the best proportion of genetic operators: elitism, crossover, and mutation, as well as mutation probability p_{mu}. Our experience shows that it is better to keep the percentage of elitism and crossover offspring small and maximize mutation with p_{mu} in [0.1, 0.5] range. For the rest of our experiments, we set portions of elitism, crossover, and mutation to be 30%, 30%, and 40% of the whole population (C_{pop} = 1000), respectively and the mutation probability p_{mu}= 0.1.

VI. Results

We compare the performance of BT and GA based on 305 randomly generated resources, each of which can be served as one of the four types of workers: Web server, DB server, CPU-oriented worker, and storage server. Our BT approach generates only one optimal solution while the GA gives a distribution of fitness values in the entire population. The boxplots in graphs present statistics (min, 25%, mean, 75%, and max) of GA results from 100 independent runs.

1) *Feasibility of GA by Fitness Evaluation*

Figure 6(a) presents the fitness value defined in Eq. 6 using four application scenarios. Compared to an optimal solution of BT, the distribution of boxplot depicts GA's solution; where at a maximum point GA can achieve the optimal fitness. This shows that both approaches can achieve optimal results, subject to running time constraints. We further study tradeoffs between performance and cost in the next section.

2) *Tradeoffs between Performance and Cost*

The distribution of performance that GA can achieve is captured in each single bar in Figure 6(b). Web and eCom show relatively large variations in performanace, while the distribution of performance in fMRI and MapR is mostly stable. For all four applications GA manages to find the optimal solution, which is equivalent to the solution found by BT. Although eCom and fMRI can achieve the maximum in performance, they must pay more monetary cost on average than the BT case as shown in Figure 6(c).

We focus on fMRI and MapR to study the performance bottleneck. We note that there exists a yellow color-coded bottleneck worker in the middle of the workflow in Figure 5(c). This bottleneck worker having 4 incoming edges as well as 4 outgoing edges can be identified by the hub-bottleneck-authority idea using Eq. 9. We also note that the overall performance for the map reduce application siginificantly depends on one DB worker in the middle of the workflow, which is a bottleneck.

The quality of results from the GA algorithm is closely tied to how well it does in assigning resources to bottleneck workers. If those workers are assigned well, the algorithm finds a global maximum. If not the algorithm may become trapped in a local maximum, because once a sub-optimal node has been selected, the algorithm must reassign many dependent workers through widespread mutation or fortuitous crossover to escape the local maximum. This observation motivates us to develop our knowledge-based approach in Section IV.3.

3) *Sensitivity of Fitness Function to GA Parameters*

We analyze two parameters of GA: generation size and population size, and how they affect the fitness function. Although we only present one application case, eCom, this idea is easily applicable to other applications. The challenge is to find a saturation point of the fitness when varying both the population size and the generation size at the same time. In Figure 6(d), when getting closer to the orange-yellow color, GA approaches the optimal fitness. This analysis allows reducing GA's runtime significantly by limiting the generation and the population size. In our experiments, we use the values found by this analysis for these two parameters.

4) *Runtime Evaluation*

The difference between the solid line (BT) and the boxplots (GA) in Figure 6(e) clearly shows that GA is roughly 2 times faster than BT. The difference between Web and MapR clearly shows this. Theoretically, the difference will become larger as the search space becomes bigger. GA has small algorithmic cost in term of search space because it has a limited population of solutions to investigate the space.

5) *Knowledge-based GA Evaluation*

We evaluate the improvement in GA performance from knowing the bottleneck locations identified by Eq 9, and assigning the worker which has the perfect match from the knowledge-base by Eq. 10. The benefit can be measured as reduction of complexity of GA. We identify one worker with the top bottleneck score, yellow color-coded nodes in fMRI (Figure 5c) and MapR (Figure 5d). In Figure 6(f), the results show that the bottleneck information can improve algorithm's running time by 27% on overage in fMRI, while the performance enhancement (23% on average) is observed in the MapR application. Using the knowledge GA, there may be possibilities to improve optimality and performance by applying further ideas. We leave this part for the future work.

VII. Conclusion

We presented an application-optimal resource allocation framework for deploying applications across multiple cloud providers. The framework is based on the idea that application designers will have varying goals with respect to multiple performance metrics (e.g., throughput, reliability, cost, etc.) and can make use of resources with different performance and cost profiles to meet those goals.

In support of this we modeled applications and resources in ways amenable to both exact solutions for small problems, (backtracking), and bounded-time heuristics for larger, more complicated problems, (genetic algorithms). We demonstrated that both algorithm classes worked well in their respective applicability domains.

We were further able to improve each system's performance based on domain-specific knowledge. Backtracking directly takes advantage of aggressive pruning of the search space when user performance metrics are amenable to it. By analyzing the workflow graphs of applications using existing graph theory algorithms, we were able to choose initial populations for the genetic algorithms that were more likely to achieve optimal results.

VIII. ACKNOWLEDGEMENT

This work has done during the Ahnn's summer internship period at ISI in 2010. It is funded by InfoSys.

REFERENCES

[1] J. R. Bitner, and E. M. Reingold, "Backtracking programming techniques," Communications of the ACM, vol. 18, pp. 651-655, 1975.

[2] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," J. Parallel Distrib. Comput., vol. 47, pp. 8-22, 1997.

[3] D. E. Goldberg, "Genetic Algorithms in search, optimization, and machine learning," Addison-Wesley, 1989.

[4] W-N Chen and J. Zhang, "An ant colony optimization approach to a grid workflow scheduling problem with various QoS requirements," IEEE Trans. Of Systems and Cybernetics, vol. 39, no. 1, pp. 29-43, 2009.

[5] H. Topcuoglu, S. Hariri, and M.-Y Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE Trans. Parallel Distrib. Syst., vol. 13, no. 3, pp. 260-274, 2002.

[6] T. D. Braun et al, "A Comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," J. Parallel Distrib. Comput., vol. 61, pp. 810-837, 2001.

[7] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using Genetic Algorithms," Scientific Programming, vol. 14, no. 3,4, pp. 217-230, 2006.

[8] Y. Xiong, N. Sadeh, and K. Sycara, "Intelligent Back-tracking techniques for job shop scheduling," In Proc. Of the 3rd Int. Conf. on Principles of Knowledge and Representation and Reasoning, pp. 14-23, 1992.

[9] V. D. Martino and M. Mililotti, "Scheduling in a grid computing environment using Genetic Algorithm," IPDPS, pp. 297, 2002.

[10] J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia, "DADL: Distributed Application Description Language," USC/ISI Technical Report # ISI-TR-664.

[11] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," Proceedings of IMC, 2010.

[12] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environments," SIAM, 1998.

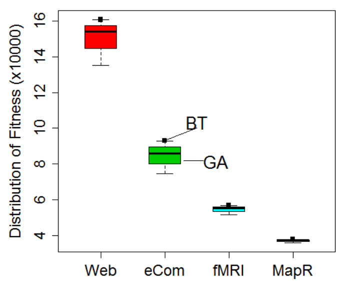
[13] H. Kanoh and Y. Sakamoto, "Knowledge-based genetic algorithm for university course timetabling problems," Journal International Journal of Knowledge-based and Intelligent Engineering Systems, 2008.

[14] S-J Ko and J-H Lee, "Optimization of Association Word Knowledge Base through Genetic Algorithm," KDD, 2002.

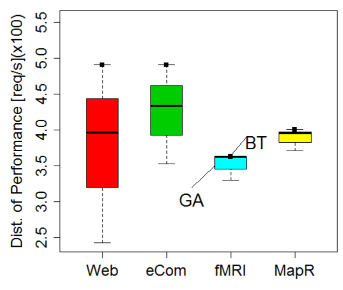
[15] H. Yanrong and S. X. Yang, "A knowledge based genetic algorithm for path planning of a mobile robot," Robotics and Automation, 2004.

[16] R. Dechter and J. Pearl, "Network-based heuristics for constraint satisfaction problems," Articial Intelligence, vol. 34, no. 1, pp. 38-73, 1988.

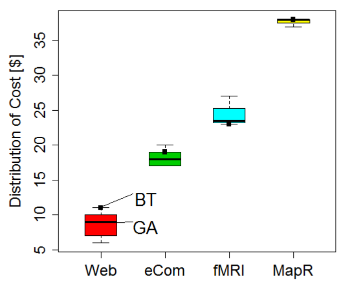
[17] Amazon EC2 (n.d.) [Online]. Available: http://aws.amazon.com/ec2/pricing/.



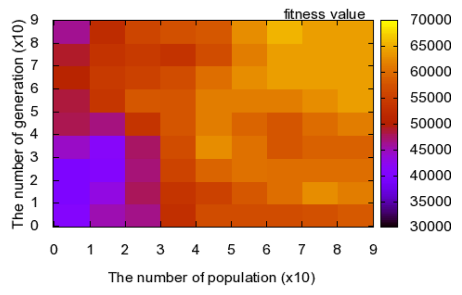
(a) Fitness



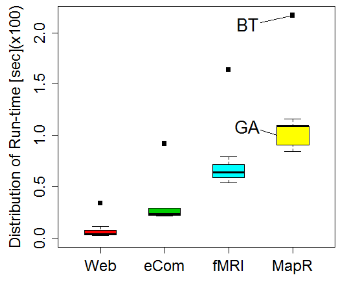
(b) Performance



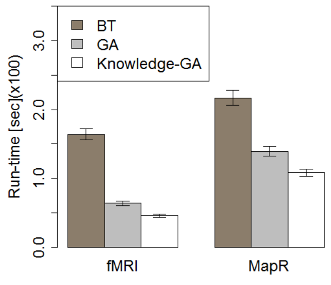
(c) Monetary Cost



(d) Sensitivity of GA Fitness



(e) Runtime of GA and BT



(f) Runtime of Knowledge-GA

Figure 6. (a) Fitness of GA; (b-c) Tradeoffs between performance and cost; (d) Sensitivity of fitness of GA to population size and generation size; (e) Comparisons of runtime between GA and BT; (f) Comparisons of average runtime between BT, GA and Knowledge-based GA with 95% confidence_intervals.