Fast (Re-)Configuration of Mixed On-demand and Spot Instance Pools for High-throughput Computing

Alexandra Vintila
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
A.A.Vintila@vu.nl

Ana-Maria Oprescu
Informatics Institute
Universiteit van Amsterdam
Amsterdam, The Netherlands
A.M.Oprescu@uva.nl

Thilo Kielmann
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
Thilo.Kielmann@vu.nl

ABSTRACT

Commercial cloud offerings let users allocate compute resources on demand, charging based on reserved time intervals. Users, however, lack guidance for assembling instance pools from different cloud instance types, in order to control completion time and monetary budget. BaTS, our budget-constrained scheduler uses tiny statistical samples of task executions in order to predict completion times (and associated costs) for given bags of tasks, allowing the user to favor either fast execution or low computation budget. BaTS' estimator, however, can not handle variably-priced spot instances appropriately.

In this work, we present a new prediction module for BaTS that quickly computes accurate approximations to the Pareto set of mixed on-demand and spot instance pools, based on a genetic algorithm (GA). This new approach allows BaTS to react to changing spot instance prices at runtime by re-configuring the instance pool according to the user's runtime and budget constraints. Simulator-based results show that the GA can approximate the Pareto sets for machine configurations in about 30 seconds time, without noticeable loss of quality, compared to an exact solution, computed offline within 40 to 60 minutes time.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; I.2.8 [**Problem Solving, Control Methods, and Search**]: Scheduling

Keywords

Infrastructure-as-a-Service, Pareto frontier, Genetic Algorithms

1. INTRODUCTION

In high-throughput computing, parameter sweep or bag-of-tasks applications are as dominant as computationally demanding. An ideal match for these demands can be seen in commercial cloud offerings, like Amazon's EC2 [6, 7], where computers can be allocated ("rented") for given time intervals. The various commercial offerings differ not only in price, but also in the types of machines that can be allocated. While all machine types are described in terms of CPU clock frequency and memory size, it is not clear at all which machine type can execute a given user application faster than others, let alone predicting which machine type can yield the best price-performance ratio. The problem of allocating the right number of machines, of the right type, for the right time frame, strongly depends on the application program, and is left to the user.

This problem is addressed by BaTS, our budget-aware scheduler for bags of tasks [13].[1] BaTS requires no a-priori information about task completion times. In a small, initial sampling phase, BaTS learns properties of the bag's runtime distribution for each considered cloud offering. BaTS provides the user with a list of makespan/cost options (with different combinations of machines) to choose from, ranging from the cheapest to the fastest offer.

Our current implementation, however, provides only a small and fixed list of possible makespan/cost options, namely the cheapest option, the cheapest plus 10 % and plus 20 % of budget, as well as the fastest option, and the fastest minus 10 % and minus 20 % of budget. This set of options might contain duplicates or obviously unattractive schedules. Instead, the Pareto set of options should be presented to the user as those are dominating all other possible machine configurations. Another drawback of BaTS's current makespan/cost prediction module is that it uses a bounded-knapsack solver tailored for price ranges of on-demand instance types, which renders it unfit to deal with the combined price range of spot and on-demand instances.

In this paper, we overcome these deficiencies of our previous BaTS implementation. We have implemented a genetic algorithm that computes approximations to the Pareto set of makespan/cost options (based on their underlying instance pool configurations). This genetic algorithm allows BaTS to quickly (within seconds) compute a set of Pareto-optimal configurations, and hence is able to quickly react to changing spot instance prices.

We have evaluated the quality of the Pareto sets identified by our genetic algorithm using our BaTS simulator [14] that we have extended by price history logs from Amazon's spot market. We compare our solutions to offline computations of the exact Pareto sets, both in terms of quality and of

[1]BaTS is implementing the *Task Farming Service* of the ConPaaS platform, available from *www.conpaas.eu*

computational effort needed. Our results show that the GA can approximate the Pareto sets for machine configurations in about 30 seconds time, without noticeable loss of quality when compared to an offline computation of an exact solution that takes in the order of 40 minutes to compute.

This paper is structured as follows. Section 2 describes the BaTS scheduler and discusses related work. Section 3 presents our new genetic algorithm for approximating Pareto sets. We evaluate its performance in Section 4. Conclusions are drawn in Section 5.

2. BACKGROUND AND RELATED WORK

Before we present our new genetic algorithm for predicting makespan/cost combinations of instance pools, we briefly summarize the basics of our BaTS scheduler, consider Pareto optimality, and discuss related work.

2.1 The BaTS Scheduler

BaTS is scheduling large bags of tasks onto multiple cloud platforms. The individual tasks are scheduled in a self-scheduling manner onto the allocated machines. An initial sampling phase computes a list of budget estimates providing the user with flexible control over budget and makespan. The execution phase allocates a number of machines from different clouds, and adapts the allocation regularly by acquiring and/or releasing machines in order to minimize the overall makespan while respecting the given budget limitation.

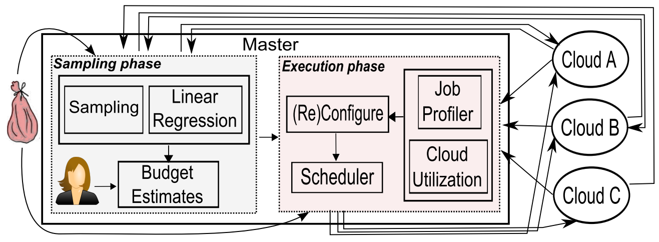

Figure 1: BaTS sampling phase (left) and execution phase (right).

Our task model needs no prior knowledge about the task execution times; just the total number of tasks. About the machines, we assume that they belong to certain categories (e.g. EC2's "Standard Large") and that all machines within a category are homogeneous. The only information BaTS uses about the machines is their (hourly) price.

In this work, we use fixed hourly prices for on-demand instances and add information about the fluctuating prices for spot instances, based on spot price history provided by Amazon. As our focus is on computing the configurations of instance pools, we leave bidding strategies for the spot instances to future work; instead we assume that BaTS will place sufficiently high bids that avoid instance preemption. Bidding strategies for spot instances merely trade chances of not being preempted against acceptable price increases. With the work presented here, we can instead quickly drop too-expensive spot instance types and request more cost-efficient ones instead.

Figure 1 sketches BaTS' overall system architecture. BaTS itself runs on a *master* machine, where the bag of tasks is available. Figure 1(left) sketches the sampling phase, where BaTS learns the bag's stochastic properties and uses linear regression to translate task completion times across clouds [13]. In the sampling phase, all machine types given by the user are tested; it is sufficient to sample a machine type only once, either under on-demand or under spot allocation: the machine type is the same, independent of the billing scheme.

BaTS generates a list of budget estimates accordingly, reflecting execution speed and profitability (price/performance ratio). The user is then asked to select one of the budgets (e.g., faster or cheaper) corresponding to a desired schedule. The user's choice then determines the machines allocated by BaTS for the execution phase, shown in Figure 1(right). Here, BaTS allocates machines from various clouds and lets the scheduler dispatch the tasks to the cloud machines. Feedback, both about task completion times and cloud utilization, is used to reconfigure the clouds periodically, as needed.

In this work, we improve on the existing budget/makespan estimation module, that is based on a bounded-knapsack solver, which provides only solutions of limited usefulness while being computationally demanding, also by the amount of memory needed to investigate the problem space. As our goal is to exploit combinations of on-demand and spot market instances, we need a new estimation module that can quickly handle the large problem space created by the spot instance prices that are both fluctuating and are an order of magnitude lower than the corresponding on-demand prices of the respective instance types.

It is worth mentioning that our very general task model does not allow any hard budget guarantees for the execution of the entire bag. Since BaTS has no a-priori information about the individual execution time of each task, we cannot guarantee that a certain budget will be definitely sufficient. The case might always occur that one or more outlier tasks, with exceptionally high completion time, might be scheduled only towards the end of the overall execution. In [14], we presented an optimization for the tail phase execution that reduces the severeness of this problem.

2.2 Pareto frontiers

In multi-criteria optimization, multiple and possibly conflicting properties of solutions need to be taken into account. Pareto introduced the concept of dominance, which simply states that if a given solution S_1 outperforms another solution S_2 for at least one objective function, while it performs equally well for the remaining objective functions, then S_1 dominates S_2 (is strictly better). The set of all non-dominated solutions is called a *Pareto frontier* or *Pareto set*.

The goal of our work is to compute the set of Pareto-optimal instance pools for a given bag of tasks, within the limitations on the numbers of available instances per type given by the cloud provider. Because computing the precise Pareto set is computationally challenging, we present a genetic algorithm that can quickly compute an approximation to the precise Pareto set.

2.3 Related Work

High-throughput computing systems are legion, dating back to Condor [10]. Our own previous work on BaTS [13] and the work in [11] focus on executing bags of tasks in a high-throughput manner on cloud platforms, while keeping both execution makespan and monetary cost under control. Here, we extend BaTS by a means to quickly estimate the

Pareto front of instance configurations, extending the applicability of BaTS to spot-market instances and combinations of on-demand and spot instances.

The ExPERT framework [3] constructs the Pareto-frontier of scheduling strategies from which it selects the one that best fulfills a user-provided utility function. Unlike ExPERT, we focus on the makespan of the execution as a whole, and we compute the Pareto front of machine (cloud instance) combinations on which BaTS executes the tasks of a bag in a self-scheduling manner.

Genetic algorithms have often been employed to solve the problem of optimal resource scheduling. There is a large body of research conducted in this area, and the existing approaches differ with respect to the chromosome encoding, crossover operations, fitness function and convergence (stop) conditions. Early work in [2] uses a genetic algorithm to schedule jobs on heterogeneous machines, optimizing only the makespan. Similar to our approach, they too start with randomly generated chromosomes (to ensure diversity) and convert invalid chromosomes to valid ones at a crossover level.

Research conducted in [16] comes closer to our work, since it targets the optimal resource scheduling problem in a cloud computing environment, while considering the overall cost. However, they have different optimization objectives: quality of service and response time for requests. They also use an integer coded chromosome type, but their fitness function uses the summation of the representative coefficients for each objective, while ours is based on multiplication.

Our work has been inspired by [15], where the authors build a genetic algorithm to approximate the Pareto frontier of schedules for jobs and data transfers in grid environments. Their method does not scale well with the number of jobs to be executed, as their gene encoding has one entry per job in each chromosome. Instead, we only encode the numbers of instances per instance type within our chromosomes, which is independent of the actual number of tasks, and hence scales to the execution of large bags.

3. GENETIC ALGORITHM

We start from a typical genetic algorithm structure (Algorithm 1): iterate over a number of generations (populations) until the algorithm converges according to a termination condition. In each iteration, new individuals are randomly generated or created through recombination and mutation. The new individuals together with an elite of the current population form an intermediate population. A second selection step chooses among the extended (intermediate) population those individuals that will become part of the new population, that is the generation of the next iteration. During the selection process, a fitness function is used to rank individuals. The result of the algorithm is the estimated Pareto front.

3.1 Chromosome and gene encodings

In genetic algorithms, a population contains a number of chromosomes (individuals), and each chromosome consists of an (equal) number of genes. Their mapping to the real-world problem widely varies from domain to domain.

In our approach, a chromosome represents a configuration of machines from different types. A gene encodes the number of machines of a certain type (including zero) and is represented as an integer. Thus, a chromosome is represented as an integer array. Table 1 shows an example of a chromosome encoding for a valid machine configuration given a pool of the following Amazon EC2 instance types: t1.micro, m1.small, m1.medium in both pricing policies: on-demand [6] and spot [7]. Therefore, we get a total of 6 genes.

Algorithm 1 Genetic Algorithm to Estimate Pareto Fronts

```
Input: machine types
Output: estimated Pareto front
 1: population [POPULATION_SIZE]
 2: for fixed number of iterations do
 3:     while population array is not full do
 4:         add random valid chromosome to population
 5:     end while
 6:     for all chrm C in population do
 7:         C.fitness ← computeFitness(C)
 8:     end for
 9:     population ← selectNewPopulation(population)
10: end for
11: buildParetoFront(population)
12: if Pareto front size < threshold then
13:     go once to line 2
14: end if
15: return the biggest Pareto front found
```

Algorithm 2 computeFitness

```
Input: chromosome C
Output: fitness value
 1: maxCost ← maximum cost found in population
 2: minCost ← minimum cost found in population
 3: diffCost ← maxCost - minCost
 4: maxMsp ← max makespan in population
 5: minMsp ← min makespan in population
 6: diffMsp ← maxMsp - minMsp
 7: // clCost - closeness to min cost (%)
 8: // clMsp - closeness to min mspan (%)
 9: clCost ← $(\text{maxCost} - \text{C.cost} + 1)/\text{diffCost}$
10: clMsp ← $(\text{maxMsp} - \text{C.msp} + 1)/\text{diffMsp}$
11: return Max(clCost, clMsp) · clCost · clMsp
```

gene1 micro	gene2 small	gene3 medium	gene4 spot.micro	gene5 spot.small	gene6 spot.medium
39	28	15	50	65	20

Table 1: Example of a chromosome containing genes for six machine types.

We have chosen this type of representation because it is more suitable for our problem than the traditional binary representation of a chromosome. By having integers as genes it is easier to compute the makespan and cost of a schedule, as well as to check the validity of the chromosome with respect to certain constraints. Each machine type has a limit on the number of machines the user can acquire. Another constraint is the limit on the overall number of machines.

3.2 Fitness function

In order to estimate the Pareto front, we need to find those individuals in the population that are closest to the Pareto front, that is the individuals which minimize both

Algorithm 3 selectNewPopulation

```
Input: population
Output: new population
 1: sort population decreasing by fitness, which was previously computed with Algorithm 2
 2: // elitism
 3: add top $p_e$% of population to intermediate population
 4: // crossover
 5: for $i = 0$ to top $p_x$% of the population do
 6:     choose different parents: P1 and P2
 7:     // by using roulette wheel selection,
 8:     // to create two children: C1 and C2
 9:     (C1, C2) ← recombination(P1, P2)
10:     add C1 and C2 to intermediate population
11: end for
12: sort intermediate population increasing by cost
13: move top $p_c$% to the new population
14: sort intermediate population increasing by makespan
15: move top $p_m$% to the new population
16: sort intermediate population increasing by msp+cost
17: move top $p_{mc}$% to the new population
18: return new population
```

Algorithm 4 recombination

```
Input: chromosomes P1, P2
Output: chromosomes C1, C2
 1: for all gene g in the chrm P1 do
 2:     maxM ← maximum number of machines for this gene
 3:     diff ← |P1g − P2g|
 4:     mg ← min(P1g, P2g)
 5:     Mg ← max(P1g, P2g)
 6:     // a>0
 7:     mNoMachines ← max(0, mg - a · diff)
 8:     MNoMachines ← min(maxM, Mg + a · diff)
 9:     C1g ← rand(mNoMachines, MNoMachines)
10:     C2g ← rand(mNoMachines, MNoMachines)
11:     // mutation
12:     flip a bit in C1g, C2g with probability $p_f$
13: end for
14: return C1, C2
```

our objective functions: cost and makespan. To estimate their closeness to the Pareto front (given that it is a-priori unknown), we use a fitness function, where intuitively the larger its value, the better the individual. The fitness value of a chromosome is (re-)computed for each iteration.

Our objective functions incur different ranges of values: we deal with time versus money. In order to handle this, we split the search space, by predefined numbers for both the cost and the makespan ranges. This results in a grid structure layout of the search space, each chromosome belonging to a cell. The numbers used for splitting determine the accuracy of the fitness function in discriminating between chromosomes and help in handling overfitting and underfitting. Our fitness function considers closeness indicators that are normalized with respect to the value ranges. That is, we use the relative cell distance of the predicted makespan of a chromosome C to the largest predicted makespan found in the current population as a *makespan indicator* (the bigger the distance, the better the chromosome). We use the relative cell distance of the chromosome's corresponding estimated cost (necessary budget) to the largest estimated cost in the current population as a *cost indicator*.

We also reward the individuals that excel in either objective function (cost or makespan) as shown in line 11 of Algorithm 2. The chromosomes which are farther situated from at least one objective's maximum values will have a higher fitness value. The effect of giving extra weight to the superior objective value of a chromosome is that the current Pareto front is slowly pushed towards the small values of both objective ranges.

Algorithm 5 Algorithm for Building Pareto Fronts from GA Population: buildParetoFront

```
Input: population Pop
Output: estimated Pareto front
 1: sort Pop increasing by makespan
 2: i ← 1
 3: add Pop[i] to the Pareto front
 4: find smallest j > i such that the Pop[j].cost < Pop[i].cost
 5: if no j is found then
 6:     return
 7: else
 8:     i ← j
 9:     continue from line 3
10: end if
```

3.3 Population selection

The population selection must ensure a trade-off between the quality and the diversity of the next generation's individuals. To that end, genetic algorithms use such mechanisms as elitist selection, crossover operations and mutations.

Our population selection is done in two main steps. First, we build the intermediate population (lines 1–11 in Algorithm 3). Next, we select the best chromosomes by cost, makespan and their sum (lines 12–17 in Algorithm 3).

We initialize the intermediate population with the top p_e percent of the fittest chromosomes within the current population. This technique is known as *elitism* in genetic algorithms.

Next, we extract a number of pairs equal to a percentage p_x from the population. Each individual is selected based on roulette wheel selection (fitness proportional selection [1]). From each pair we obtain two children using blend crossover (BLX-a Crossover [8]), as shown in Algorithm 4, lines 1–10.

We chose this recombination technique as it proved to give the lowest number of invalid children chromosomes. The choice was made based on experiments using other types of classical recombinations adapted to our chromosome type. We tried using one point crossover and two point crossover operations, with the points of crossover between genes (number of machines). The result was that up to a half of the children were invalid, because the total number of machines allowed was exceeded, and this was unacceptable. BLX-a is the best approach because it generates a controllable amount of values within the bounds of the parent genes, which already respect all the constraints. Therefore, the probability of building an invalid child may be easily adjusted.

We are also interested in children that lie just outside the parents gene range, but within the allowed number of machines per type. Such children increase the diversity of the population and may be the key to a finding better performing machine configurations.

We ensure all created children are valid by properly addressing all corner cases (e.g. equal parents genes). A numerical example of the BLX-a crossover is shown in Figure 2. Here, parent 1 contains 10 machines of a certain type M, while parent 2 has 38 machines of the same type. Assuming $a = 0.3$ and $maxM = 40$, according to lines 7 and 8 in Algorithm 4, the pool of genes for their children will consist of all integers between 2 and 40, from which we choose randomly.

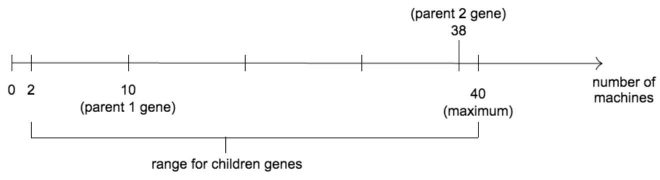

Figure 2: A numerical example of BLX-a recombination (with a=0.3) for a machine of type M.

We apply the mutation operation on each new child gene with a certain probability, p_f. The mutation consists of flipping a random bit in the integer representing the gene (Algorithm 4 line 12).

The second selection step takes as input the intermediate population and returns the next generation. This step is an optimization of the classical genetic algorithm and has the purpose of selecting, based on their cost and budget objectives, the top machine configurations for the next population (ensuring *quality*). A small percentage (100-p_c-p_m-p_{mc}) of the chromosomes with the lowest fitness in the intermediate population is always left behind, in order to make room for new randomly generated individuals (ensuring *diversity*). As a consequence, we make sure that the algorithm explores the state space.

3.4 Pareto Front

From the population resulted after the genetic algorithm's iterations, we build the Pareto front, using Algorithm 5. The number of schedules in the set is lower than the one of the schedules found in the real Pareto front. More about the size of the estimated Pareto Front compared to the real Pareto front in Section 4.5.

We use a simple heuristic to control the size of the estimated Pareto front (line 12 in Algorithm 1): if the size is below a certain (fixed) threshold, we re-execute the genetic algorithm with the same number of iterations. We considered the fixed threshold as a minimum number of chromosomes which we would like to have in the set. They represent a minimum number of configurations from which the user has to choose and which we have set to 15. In the case that the size is below this threshold, we take the Pareto front with the maximum number of elements we have found so far. The probability of such a situation occurring is very low and it decreases as the size of the population grows.

4. EVALUATION

Our algorithm approximates the Pareto front by constructing a set of (locally) Pareto-optimal solutions. Given the nature of a genetic algorithm, we cannot guarantee that we generate the entire feasible space of the optimization problem, and therefore the Pareto optimal solutions obtained by our approach may be dominated by solutions outside the considered feasible subspace. Another consequence of using a feasible subspace is a (possibly) incomplete Pareto front. We addressed these limitations with a set of heuristics applied during the recombination, mutation and selection steps, next to a tailored termination condition. Therefore, we evaluate our genetic algorithm for Pareto set approximation with respect to both its optimality and its size by comparison to the corresponding real Pareto set.

Next, we evaluate the runtime performance of the genetic algorithm compared to the exhaustive search used to compute the real Pareto set offline. We analyze these aspects on several workloads that have been found representative for real-world bag-of-tasks applications [14].

4.1 Simulated mixes of cloud instances

We simulate two different types of instance mixes, inspired by behavior of Amazon EC2, which are relevant for real-world scenarios:

(a) An equal limit on all instance types, where the user has access to a pool of in total 100 cloud instances with an upper bound of 20 for each instance type, referred to as **20-100**;

(b) Different limits for on-demand instance types versus spot-instances, where the user has access to a pool of maximum 100 cloud instances with an upper bound of 40 for each on-demand instance type and 60 for each spot instance type, referred to as **40-60-100**.

Each on-demand instance type has an associated price and (simulated) execution speed. For each on-demand instance type we simulate a corresponding spot instance type, with the same characteristics, but a different (bidding) price. In total, we use six instance types:

- **t1.micro** - executes tasks according to their generated runtime and costs \$0.020 per hour;
- **m1.small** - executes tasks twice as fast as "t1.micro" and costs \$0.065 per hour;
- **m1.medium** - executes the tasks three times as fast as "m1.small" and costs \$0.130 per hour;
- **spot t1.micro** - same speed as "t1.micro" and costs \$0.003 per hour;
- **spot m1.small** - same speed as "m1.small" and costs \$0.007 per hour;
- **spot m1.medium** - same speed as "m1.medium" and costs \$0.013 per hour;

4.2 Workloads

We evaluate our genetic algorithm on three different kinds of workloads, modeled after real bags of tasks, using a normal distribution, a (heavy-tailed) Levy-truncated distribution, and a mixture of distributions. Each workload contains 1000 tasks, with runtimes generated according to the respective distribution type.

Normal distribution has been identified as relevant for bag-of-tasks workloads by research conducted in [9]. Accordingly, we have generated a workload following the normal distribution $N(15, \sigma^2), \sigma = \sqrt{5}$ (in minutes, see Fig. 3), abbreviated as **NDB** ("normal distribution bag").

Research based on large traces [12] shows that some bags of tasks have a skewed distribution, bounded by some maximum value. To model such bags, we generate workloads according to a truncated Levy distribution with a scaling factor (τ) of 12 minutes and a maximum value (b) of 45 minutes, as shown in Fig. 3, abbreviated as **LTB** ("Levy-truncated distribution bag").

Another real application was provided by The First International Data Analysis Challenge for Finding Supernovae (DACH) [4]. The task runtimes depicted in Fig. 3 were obtained by running the entire workload on a reference machine. We model this workload as a mixture of distributions, generating workloads of which task runtimes exhibit a root-mean square deviation from the real workload of less than 5%. The mixture uses a combination of two Cauchy distributions (to mimick the "hunches" of the real workload) and three uniform distributions. These two Cauchy distributions also account for the two *modes* of the distribution and we will refer to the corresponding workload interchangeably as a *multi-modal* or *mixture* distribution bag. Note that the tail of the real workload is actually a combination of two uniform distributions and we reflect that in our mixture. The weights of the mixture have been found by solving the respective set of equations involving first, second and third moment of each distribution. We refer to the corresponding workload as a *multi-modal* distribution bag, abbreviated as **MDB** ("multi-modal distribution bag").

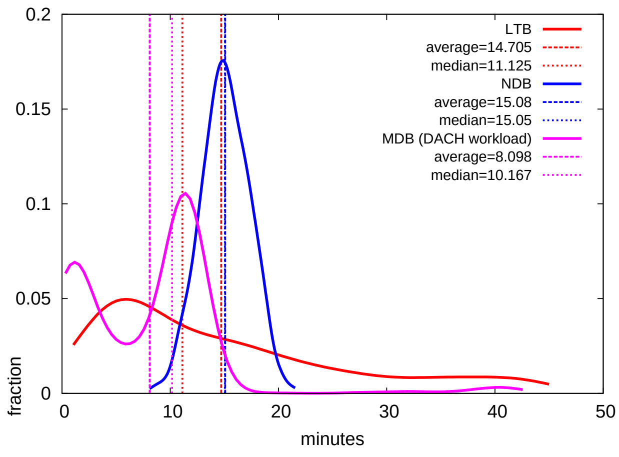

Figure 3: Distribution types used for workloads generation: normal distribution (NDB) with an expectation of 15 min., Levy-truncated (LTB) with an upper bound of 45 min. and a real-world multi-modal distribution (MDB).

4.3 Generation of Real Pareto Front

To evaluate the estimated Pareto fronts resulted from our genetic algorithm, we have implemented an exhaustive-search algorithm to generate the real (precise) Pareto front. This algorithm explores the state (feasibility) space using an optimized backtracking approach.

We start building the Pareto front with the first two generated schedules and progressively extend and improve the set. We reduce the state space by pruning the tree which has as a root an invalid machine configuration. Newly generated machine configurations are added to the Pareto front if they are better than any of those already in the Pareto set. In this way, we avoid keeping in memory all the state space and build the Pareto front by iterating through it, as we do for computing the estimated Pareto front from the population space. The estimated Pareto front allows this approach because the popualtion size is fixed and there is no memory overflow, as it would have been the case if we have used the same approach for the real Pareto front generation.

4.4 Genetic Algorithm parameters

We use two different population sizes (`POPULATION_SIZE`): 1000 (**1k**) and 2000 (**2k**). The percentage of population selected by elitism (p_e) is set to 20%. When performing the crossover operation, we extract a number of pairs equal to a percentage, p_x set to 40%, from the population. The percentages of the intermediate population selected by cost (p_c) or by makespan (p_m) are set to 20%. The percentage of the intermediate population selected by a function of cost *and* makespan (p_{mc}) is set to 50%. For each gene, the probability of mutation, p_f, is 1/15000. The value of parameter a from the BLX-a crossover is 0.3.

4.5 Experimental Results

First, we evaluate the quality of the estimated Pareto set (PS) from an optimality perspective, that is how close are our Pareto optimal solutions to the real ones. We perform two sets of experiments, one for each mix of cloud resources. We estimate the Pareto set for an instance of each workload type and we also compute the respective real Pareto set. Moreover, we evaluate the impact of the workload type on the accuracy of the estimated Pareto front. To that end, we executed our genetic algorithm with two different population sizes for the same instance of each workload type: 1000(**1k**) and 2000(**2k**).

All the results are presented in a log-log scale in Figures 4 (**NDB**), 5 (**MDB**) and 6 (**LTB**). The figures show our algorithm follows very closely the real Pareto front in every scenario. The main advantage of using a larger population is that our Pareto optimal points become sufficiently scattered across the real Pareto front, providing a proper range of relevant choices for the user.

Next, we evaluate the quality of the estimated PS with respect to the length of the real Pareto front, obtained via an exhaustive (offline) approach. In Table 2 we summarize the results obtained by running each algorithm on ten different bag instances of each workload type in scenario **20-100**. The genetic algorithm used a population of 2000 (**2k**). We repeated the experiment for scenario **40-60-100** and summarized the results in Table 3. The size of the estimated Pareto set represents the number of configurations from which the user has to choose. First of all, a large number of elements in the estimated Pareto set would lead to an overhead for the user, who would have to consider too many options for a reasonable amount of time (e.g.: 1 min), this situation being against our purpose. Moreover, the majority of the configurations in the real Pareto front have a very small difference in time and cost. In this way, a group of configurations from the real Pareto set can be considered as approximated to the nearest configuration in the estimated Pareto set. Though the real Pareto front is richer in quantity, we have already shown in Figures 4, 5 and 6 that our estimated Pareto front is similar in quality, providing a sufficient number of Pareto optimal solutions. Currently, we manually investigate and

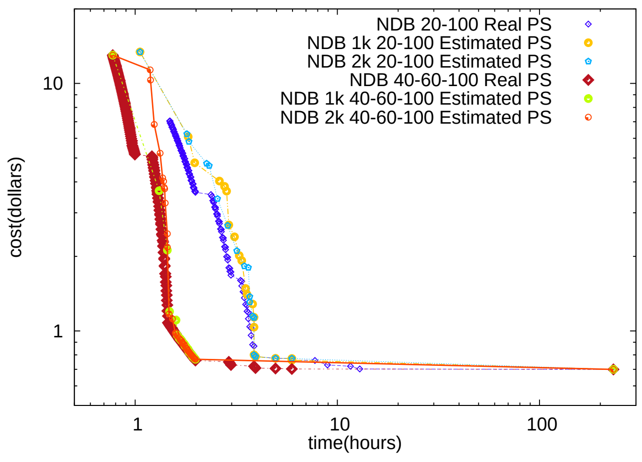

Figure 4: NDB bag, real Pareto set (PS) versus estimated Pareto set for pool mixes of type 20-100 and 40-60-100 , when using two different population sizes: 1000 (1k) and 2000 (2k). To enhance visibility, the origin is set at 0.5 and 0.5.

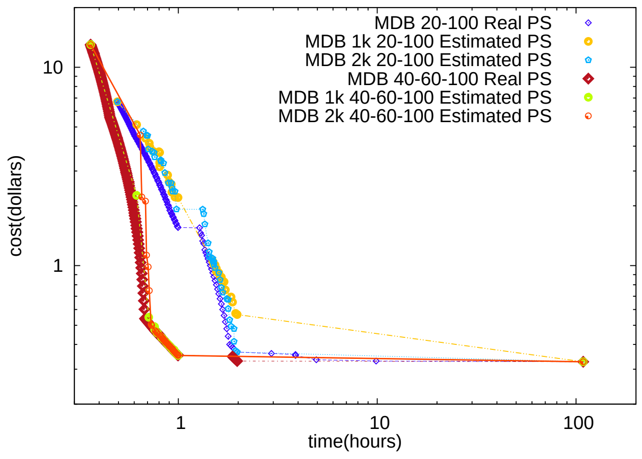

Figure 5: MDB bag, real Pareto set (PS) versus estimated Pareto set for pool mixes of type 20-100 and 40-60-100 , when using two different population sizes: 1000 (1k) and 2000 (2k). To enhance visibility, the origin is set at 0.3 and 0.2.

assess the estimated Pareto fronts. However, one direction followed by work in progress focuses on implementing automatized quality assessment techniques.

Table 2: Size of real and estimated Pareto sets on a mix of type 20-100, population size 2k

Bag Type	Exhaustive Search			Genetic Algorithm		
	Avg	Min	Max	Avg	Min	Max
NDB	124.6	84	181	28.7	17	41
MDB	157.7	144	184	35.5	31	39
LTB	121.8	93	201	26.3	19	35

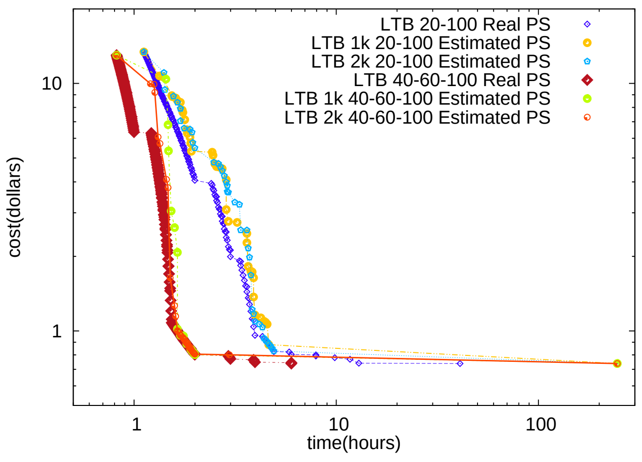

Figure 6: LTB bag, real Pareto set (PS) versus estimated Pareto set for pool mixes of type 20-100 and 40-60-100 , when using two different population sizes: 1000 (1k) and 2000 (2k). To enhance visibility, the origin is set at 0.5 and 0.5.

Table 3: Size of real and estimated Pareto sets on a mix of type 40-60-100, population size 2k

Bag Type	Exhaustive Search			Genetic Algorithm		
	Avg	Min	Max	Avg	Min	Max
NDB	188.0	155	204	42.7	38	50
MDB	247.9	196	278	31.5	18	44
LTB	193.3	162	214	39.5	18	54

We use the same set of experiments to evaluate the runtime performance of our algorithm against the exhaustive (offline) approach. All experiments were performed on compute nodes (dual-quad-core 2.4 GHz CPU configuration and 24GB memory) of our DAS4 system [5]. The corresponding results are collected in Tables 4 and 5. As the problem size increases, our results show that the genetic algorithm categorically outperforms the exhaustive-search approach, reducing the time-to-solution from 40–60 minutes to about 30 seconds, which enables online reconfiguration of instance pools, for example in case of spot price changes.

Table 4: Runtimes (sec) for computing real and estimated Pareto sets on a mix of type 20-100, population size 2k

Bag Type	Exhaustive Search			Genetic Algorithm		
	Avg	Min	Max	Avg	Min	Max
NDB	43.5	10	100	30.5	30	31
MDB	72.2	48	84	30.6	30	31
LTB	27.2	12	76	30.6	30	31

5. CONCLUSIONS

In previous work, we have demonstrated the efficacy of our BaTS scheduler for large bags of tasks on multiple cloud environments [13, 14]. Without any a-priori knowledge of task runtimes, BaTS uses a tiny sample of tasks for estimating the price-performance ratios of the available types of cloud

Table 5: Runtimes (sec) for computing real and estimated Pareto sets on a mix of type 40-60-100, population size 2k

Bag Type	Exhaustive Search			Genetic Algorithm		
	Avg	Min	Max	Avg	Min	Max
NDB	2748.0	1579	4207	31.1	31	32
MDB	3655.0	1937	5798	31.0	30	32
LTB	2500.3	1656	3826	31.1	31	32

instances, for the given bag. Based on these ratios, BaTS computes a set of estimated makespan-cost alternatives for executing the bag, based on different instance pools that are composed of the available instance types. The user can then select one of these alternatives, expressing his or her preference for either cost efficiency or execution speed. During the bulk execution of the bag, BaTS then monitors the execution and adjusts, if necessary, the selected instance pool. This approach works pretty well, even though there are no hard guarantees (only stochastic properties) that the execution schedule will stick to the estimated limits [14].

In this work, we overcome one weakness of BaTS, that is the quality of the presented makespan-cost alternatives. We replace the original, static set of alternatives by an approximation to the set of Pareto-optimal solutions, yielding a (mostly) complete set of useful instance type combinations. As this computation is very demanding, both in terms of computation complexity and memory requirements, we have implemented a genetic algorithm that computes an approximation to the precise Pareto set.

Our experiments show that computing the precise Pareto set takes in the order of 40 to 60 minutes on a regular compute node, which is only useful for offline computations. Our GA, however, approximates the exact Pareto set in about 30 seconds, which allows for quick re-configuration of instance pools at runtime, for example when using spot market instances that face fluctuations of their hourly prices (and hence price-performance ratios).

We have evaluated the quality of our GA-based approach using the simulator introduced in [14]. Based on bags with three different task runtime distributions (normal, heavy tailed, and a multi-modal mixture of distributions), we have shown that our GA computes approximations that deviate only marginally from the precise Pareto sets, and does so within seconds, enabling its use for fast and efficient reconfigurations at runtime.

We are currently integrating our GA-based approach into the BaTS implementation that is part of the ConPaaS platform, enabling the use of spot market instances, next to providing a large choice of Pareto-optimal solutions. Necessary for this integration is further research into bidding strategies for spot instances that provide a trade-off between cost-efficiency and availability probabilities of the instances. Our new and fast GA-based estimator will enable such computations at runtime.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement FP7-ICT-257438 (Contrail).

6. REFERENCES

[1] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, Oxford, UK, 1996.

[2] F.L. Beck and C.S. Thomalla. A genetic algorithm for realistic resource scheduling. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 4, pages 2522–2527 vol.4, 2001.

[3] O. Agmon Ben-Yehuda, A. Schuster, A. Sharov, M. Silberstein, and A. Iosup. Expert: Pareto-efficient task replication on grids and a cloud. In *IPDPS'12*, 2012.

[4] The First International Data Analysis Challenge for Finding Supernovae. http://www.cluster2008.org/challenge/. held in conjunction with IEEE Cluster/Grid 2008.

[5] The Distributed ASCI Supercomputer (DAS4). `http://www.cs.vu.nl/das4/`.

[6] Amazon EC2 Instance Types. `http://aws.amazon.com/ec2/instance-types/`.

[7] Scientific Computing Using Spot Instances. `http://aws.amazon.com/ec2/spot-and-science/`.

[8] Tomasz Dominik Gwiazda. *Genetic Algorithms Reference.* Tomasz Gwiazda, 2006.

[9] Alexandru Iosup, Omer Ozan Sonmez, Shanny Anoep, and Dick H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *HPDC*, 2008.

[10] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988.

[11] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid 2010*, 2010.

[12] Tran Ngoc Minh, Lex Wolters, and Dick Epema. A realistic integrated model of parallel system workloads. In *CCGRID'10*, 2010.

[13] Ana-Maria Oprescu, Thilo Kielmann, and Haralambie Leahu. Budget estimation and control for bag-of-tasks scheduling in clouds. *Parallel Processing Letters*, 21(2):219–243, 2011.

[14] Ana-Maria Oprescu, Thilo Kielmann, and Haralambie Leahu. Stochastic Tail-Phase Optimization for Bag-of-Tasks Execution in Clouds. In *5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2012)*, Chicago, IL, USA, November 2012.

[15] Javid Taheri, Albert Y. Zomaya, and Samee U. Khan. Genetic algorithm in finding pareto frontier of optimizing data transfer versus job execution in grids. *Concurrency and Computation: Practice and Experience*, 2012.

[16] Yibin Wei and Ling Tian. Research on cloud design resources scheduling based on genetic algorithm. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 2651–2656, 2012.