2010 12th IEEE International Conference on High Performance Computing and Communications

Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market

Michael Mattess, Christian Vecchiola, and Rajkumar Buyya
Cloud Computing and Distributed Computing (CLOUDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{mmattess, csve, raj}@csse.unimelb.edu.au

Abstract

Dedicated computing clusters are typically sized based on an expected average workload over a period of years, rather than on peak workloads, which might exist for relatively short times of weeks or months. Recent work has proposed temporarily adding capacity to dedicated clusters during peak periods, by purchasing additional resources from Infrastructure as a Service (IaaS) providers such as Amazon's EC2. In this paper, we consider the economics of purchasing such resources by taking advantage of new opportunities offered for renting virtual infrastructure such as the spot pricing model introduced by Amazon. Furthermore, we define different provisioning policies and investigate the use of spot instances compared to normal instances in terms of cost savings and total breach time of tasks in the queue.

1 Introduction

Many organizations rely on dedicated clusters for running computing applications. Users submit applications as Bags-of-Tasks to a scheduler, which then executes those tasks as independent processes. A typical cluster purchase is based on expected usage and funding. The size of the cluster and therefore the number of available resources is static, and cannot increase dynamically. It is common to have periods when the demand on the cluster exceed capacity but these peaks do not justify the purchase of new resources to increase the throughput. Nonetheless, over shorter time periods the increased load can be very heavy, and making the existing IT infrastructure inadequate to provide a reasonable service. In these cases, the ability of temporarily extending the capacity of the cluster by renting resources from an external provider can be a viable proposition. Such an opportunity is offered by Cloud Computing [1][4], and more precisely Infrastructure as a Service (IaaS) providers, which, by leveraging virtual machine technology, deliver IT infrastructure on demand on a pay per use basis. Organizations can then integrate external compute resources into their existing systems. However, since the use of resources from Cloud providers is charged as a utility, a careful investigation on how and when to leverage them is necessary.

In this work we focus on temporarily extending a local cluster through the use of resources leased from an IaaS provider to specifically allow a statically sized system to cope with an increased load. We consider this from the point of view of the cluster operator, who wishes to provide a generally reasonable level of service in terms of the amount of time tasks spend in the queue while also aiming to minimize expenditure. For this purpose we present two scheduling policies controlling the provisioning of virtual resources and their release that dynamically react to the changes in the task queue. Further policies are presented to make use of Amazon's recently introduced variable Spot Pricing[1] to reduce spending. As such the scope of this paper is based on how to best utilise actual commercial services available today rather than proposing new mechanisms.

The main contributions of this work are: (1) a system for dynamically growing the number of resources to handle demand using policies that minimize cost through spot market resources and (2) the evaluation of the approach using real trace data for both the workload and spot market.

The remainder of this paper is organized as follows. In Section 2 we provide background on Cloud computing and in Section 3 describe the policies used. Section 4 presents the experimental setup and metrics for the reported results in Section 5. Conclusions are presented in Section 6.

[1] http://aws.amazon.com/ec2/spot-instances/

978-0-7695-4214-0/10 $26.00 © 2010 IEEE
DOI 10.1109/HPCC.2010.77

2 Background and Related Work

Previous work has shown how commercial providers can be used for scientific applications. Deelman *et al.* [6] evaluated the cost of using Amazon EC2 and S3 services to serve the resource requirements of a scientific application. Palankar *et al.*[9] highlighted that users can benefit from mixing Cloud and Grid infrastructure by performing costly data operations on grid resources while utilizing the data availability by the Cloud.

Our work is based on a similar previous work [5] that evaluates different strategies for extending the capacity of local clusters with commercial providers. These strategies aim to schedule reservations for resource requests. Here, we consider tasks that requires a single resource to be executed.

Several load sharing mechanisms have been investigated in the distributed systems realm. Iosup *et al.* [7] proposed a matchmaking mechanism for enabling resource sharing across computational Grids. Wang and Morris [12] investigated different strategies for load sharing across computers in a local area network. Surana *et al.* [10] addressed the load balancing in DHT-based P2P networks. Balazinska *et al.* [2] proposed a mechanism for migrating stream processing operators in a federated system.

Market-based resource allocation mechanisms for large-scale distributed systems have been investigated [13][3]. In this work, we do not explore a market-based mechanism as we rely on utilizing resources from a Cloud provider that has cost structures in place. However we do make use of the market driven Spot Price of resources, but instead use actual price data rather than a hypothetical market.

2.1 Amazon EC2 Operation Model

Amazon's Elastic Compute Cloud (EC2)[2] allows users to deploy virtual machines on Amazon's infrastructure, which is composed of several data centers located around the world. When requesting a normal instance, billing is in one hour blocks and any partially used block will incur the same charge as a full block. Aside from failures in Amazon's systems an instance will keep running until it is explicitly terminated by the user. In December 2009 Amazon introduced Spot Instances, which are generally available at a significant discount. When requesting to use Spot Instances the user must provide a bid, which is the maximum price that the user is willing to pay per hour. If the spot price goes above the user's bid the spot instances will be terminated without warning. The spot price is controlled by Amazon.

The cost of a spot instance is also calculated based on one hour time blocks. However since the Spot Price varies, the hourly unit price is the Spot Price at the start of each hour. For example a spot instance is requested when the Spot Price is USD$0.03. The user will be charged USD$0.03 for the first one hour block; 30 minutes later the price changes to USD$0.05 but the user is unaffected until the second one hour block, which will then cost USD$0.05.

As with regular instances, if the user terminates the instance any partially used time blocks will be charged as a full block. However if Amazon terminates an instance due to a rise in the spot price no charges will be incurred for the partial time block. The user only pays for completed one hour time blocks if Amazon terminates the instance.

[2]http://aws.amazon.com/ec2/

3 Resource Provisioning Policies

3.1 Scenario

In this paper we are considering the scenario where a local cluster is at times overloaded, and, in an effort to maintain reasonable task queue times, additional resources are provisioned from an IaaS provider. What it means for a task queue time to be 'reasonable' will depend on the individual circumstances, however for this work we consider it to be relative to the execution time of the task itself. As an accurate execution time is not known when the task is queued, the time block requested by the user is used instead. So for a task to spend thirty minutes in the queue may be considered reasonable if it could run for up to an hour, however that queue time is not reasonable for another task for which only twenty minutes were requested.

As we are looking at executing tasks on external resources acquired from an IaaS provider, the workload is restricted to only Bag-of-Task applications. Such tasks are self-contained units of work and do not rely on communicating directly with other executing tasks for synchronization. Hence they are better suited to be executed in an IaaS environment where the networking between the nodes is generally not as capable as that in a dedicated scientific cluster.

Provisioning policies control the acquisition and release of external cloud resources. In the case of peak demands, new resources can be provisioned to address the current need, but then used only for a fraction of the block for which they will be billed. Terminating these resources just after their usage could potentially lead to a waste of money; it makes sense that they are reused in the short term before the time block expires. In order to accomplish this, we introduced a resource pool that keeps resources active and ready to be used until the time block expires.

Figure 1 describes the interaction between the scheduling algorithm, the provisioning policy, and the resource pool. Arriving tasks are placed in a queue from which they are taken and assigned to a free resource. A set of different events such as the arrival of a new task, its completion, or a timer can trigger the provisioning policy that will decide

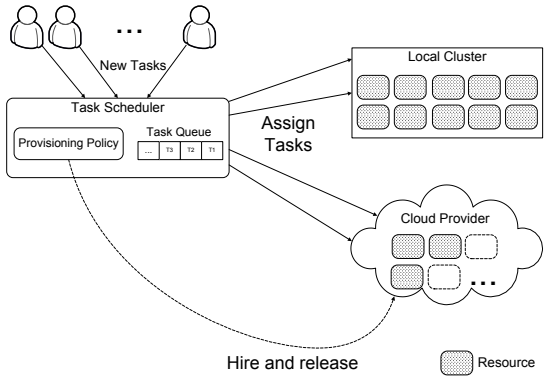


Figure 1. The task scheduling and resource provisioning scenario.

whether to request additional resources from the pool or release them. The pool will service these requests, firstly with already active resources, and if needed by provisioning new ones from the external provider.

The information about tasks, which is available to the provisioning algorithm is limited to the wall-clock time duration requested by the user for the execution of the task. As this user provided value is an upper bound on the executions time and generally significantly overestimate the actual execution time, we use a *Workload Multiplier*. This parameter is used to scale down the requested execution time, which is then used as the *Expected Run Time* of the task.

The requested time is also used to derive an 'acceptable' maximum time for the task to spend in the queue, which is given by the *Target Ratio* between the requested time and the *Maximum Queue Time*, (Equation 1). For very small tasks the *Maximum Queue Time* could become correspondingly very short, hence a lower bound of five minutes is applied to the *Maximum Queue Time*. The time by which a task must be assigned as not to breach its *Maximum Queue Time* will be is its *deadline*.

$$MaxQueTime = RequestedTime \times TargetRatio \quad (1)$$

Even though Cloud computing claims to provide vast scalability and potential for limitless growth, in reality limitations do apply. For example, Amazon EC2 initially limits each account to 20 instances in each region [3]. Hence we also impose a limit on the number of external resources that can be used at any one time.

Below are the descriptions of each of the algorithms used, Table 1 also includes an overview of the algorithms and their of behaviour.

[3] http://aws.amazon.com/ec2/faqs/

3.2 Base Provisioning Algorithm

This algorithm forms the basis of all the approaches explored in this paper. It provides the basic mechanics for requesting and releasing additional resources. At its core is a simple heuristic for predicting breaches of the *Maximum Queue Time*. This heuristic is used both when tasks are added to the queue and when a provisioned resource becomes free. The fundamental approach of the heuristic is to use the *Expected Run Time* of each task to predict when resources would become available for the next task in the queue. The heuristic first builds an array with the time each resource is expected to become available based on the *Expected Run Time* of the task currently executing on each resource. Resources that have been requested but have not yet come online are also included in the array and considered to be already available. Once the array is populated the heuristic iterates through the queue of tasks keeping track of when each resource is expected to be available again as the tasks are assigned to the next available resource. When the predicted assignment time of a task would breach its *Maximum Queue Time* the heuristic stops and indicates that a breach was detected.

When a task arrives and no resources are available it is inserted into the queue of waiting tasks. The tasks in the queue are sorted soonest *deadline* first. Once the new task is queued the breach detection heuristic is executed to determine if breaches of the *Maximum Queue Time* are expected. If the heuristic indicates that a breach will occur, one additional resource is requested unless the maximum number of resources has been reached.

When a task completes on a provisioned resource the breach detection heuristic is used to test if removing the resource is expected to lead to breaches of the *Maximum Queue Time*. If a breach is expected the resource remains in use and the next task from the queue is assigned. If on the other hand, no breach is predicted then the resource could be released back to the pool. However, before releasing the resource the task queue is searched for the task that would best make use of the remaining time block of the resource by looking at the requested time of the task. The best fit task is removed from the queue and assigned to the resource.

3.3 Base Hard

This is a variation on the *Base* algorithm, it not only relies on attempting to predict breaches of the *Maximum Queue Time* to trigger a request for additional resources, but it also regularly checks the tasks in the queue if they are about to breach the *Maximum Queue Time*. The checks are performed at a one minute interval and each task that is

Algorithm	Breach Prediction	Regular Checking	Use Spot Resources
Base	Yes	No	No
Base Hard	Yes	Yes	No
Spot Base	Yes	No	Yes
Spot Base Hard	Yes	Yes	Yes
Spot Aggressive	Yes, *Workload Multiplier 1* with spot resources.	No	Yes
Spot OnlyHard	Yes	Only with spot resources	Yes
Pure Spot	Yes	No	Yes, no retail resources used

Table 1. Overview of algorithms

within four minutes of breaching its *Maximum Queue Time* will trigger one additional resource to be requested. A four minute threshold was chosen in this case due to the configuration of the simulation, in particular the time a resource requires to boot. As described in section 4 when a resource is requested it will take three minutes for the resource to become available. Hence with a four minute threshold and checks every one minute, it should be possible to request a new resource to be available just in time to prevent any *Maximum Queue Time* breaches.

3.4 Spot Instances Algorithms

Spot Base and Spot Hard: This is the simplest approach to attempt to make use of the discounts available via spot instances. The general behaviours of the algorithms remains the same, but if spot resources are available, that is, the current Spot Price is below the configured maximum then spot resources will be requested instead of normal retail priced resources. In the event of the Spot Price rising above the maximum bid price all spot resources will be terminated by the IaaS provider as described above. If any tasks were executing on these resources they will have to be restarted on another resource and hence inserted in the queue again. The action of inserting these tasks triggers the same reaction as new tasks arriving and being inserted into the queue. Hence it may result in retail resources being requested as the Spot Price is now too high and only retail resources are available.

Spot Aggressive: When spot resources are available this algorithm uses a *Workload Multiplier* of 1. Meaning that it is more aggressive at requesting resources when cheaper spot resources are available.

Spot Only Hard: This algorithm also changes its behaviour depending on the availability of spot resources. When the Spot Price is too high and above the configured maximum bid price then this algorithm acts the same as the *Base* algorithm. However when the Spot Price falls below the maximum bid price, it begins to perform the regular checks for imminent *Max Queue Time* breaches and hence acts the same as the *Base Hard* algorithm but requesting spot resources.

Pure Spot: This algorithm only uses spot resources, hence when the Spot Price is above the configured maximum bid price no provisioning occurs. When the Spot Price is low enough to permit the usage of spot resources then it behaves like the *Base* algorithm. Whilst spot resources are too expensive the algorithm keeps count of the number of tasks that have arrived, such that when the Spot Price falls and spot resources can be used the algorithm can run the breach detection heuristic for each of these tasks that have arrived.

4 Performance Analysis - Setup

In order to evaluate each policy we built a discrete-event simulator and identified performance metrics that will drive the discussion of the experimental results. The Simulator is implemented to reflect the behaviour of Amazon EC2. It was configured with a resource 'boot' time of three minutes and to allow a maximum of 200 external resources to be used at any one time. The default limit[4] Amazon imposes on spot instances is 200 although in our simulation this limit is applied to both regular and spot resources. The local cluster was set to contain 10 resources. With the workload used this averages to 9 hours of computation time per day, which we considered a reasonable level of load on the system. Hence, the total number of resources in use could grow from 10 to 210, giving over a 20 times increase. For the simulation all resources were considered to have the same performance.

The main values used for evaluating the performance of the algorithms are the cost of renting the additional resources and the sum of the time tasks have spent in the queue beyond their *Maximum Queue Time*, which we termed the *Total Breach Time*. The average time all tasks have spent in the queue is also used at times to give an indication of the overall impact some algorithms may have

[4] http://aws.amazon.com/ec2/faqs/

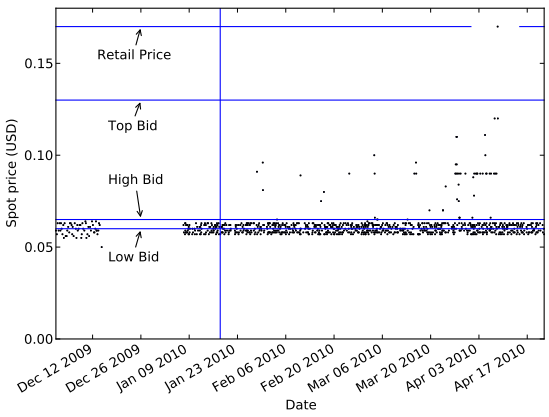


Figure 2. Spot Price data used for simulation

beyond just meeting deadlines.

To calculate a cost for the use of cloud resources we use a High-CPU Medium EC2 Instance as a reference point. We chose this instance type as it provides the best value in terms of the cost per EC2 compute unit and as such should be well suited for CPU intensive scientific applications. The true cost of using cloud resources includes other costs such as network activity both at the IaaS provider's and at the local end as well as data storage in the cloud. We have not attempted to establish these costs as they are dependent on the individual case. Hence, the cost for using cloud resources in this work is solely based on the amount of billable time.

4.1 Spot Price Trace

The trace data we are using for the Spot Prices comes from Amazon EC2 Spot Prices. The data was obtained from the published web services API for the Spot Price history. Figure 2 gives the full Spot Price history that is available for the *High CPU Medium* instance type. This spot price history is representative of Linux instances types. The figure has a point every time a new price was set. The top horizontal line in this graphs is drawn at the retail price of USD$0.17.

From this graph it is apparent that the early spot prices, prior to mid-January 2010 do not seem to be completely in line with the later pricing pattern. The gap in the graph over the Christmas period indicates that the *Spot Price* did not change during that time. Potentially the lack of price fluctuation over this period is the result of a lack in demand as the service had only been introduced earlier in the month. It is also clear that from mid January onwards the vast majority of price fluctuations are confined to a relatively narrow band between 30% and 40% of the retail price. It should however be noted that spot prices have reached the retail price in several occasions. Hence, placing a bid using the retail price there is not guarantee that a spot price will not be terminated. However, bidding above the band of spot prices will dramatically reduce the chance of a instance being terminated.

The trace data used for the simulations is that of the *High CPU Medium* instance type. The trace is trimmed to begin on Monday the 18th of January 2010 (vertical line in Figure Figure 2) to avoid the inactivity over the Christmas period. The top horizontal line in this graph is the retail price at USD$0.17, the lines below are the price points used during the simulation. They are the *Low* level located in the band of price fluctuations at USD$0.06; immediately above the band is the *High* level at USD$0.065 but still below some of the spot price points; and finally, the *Top* level is at USD$0.13 which is greater than all but a single spot price point in the trace. The *Low* level is set roughly in the middle of the band of price fluctuations, such that most instances of a new Spot Price being set will cross this threshold. On average a new Spot Price is set approximately every three hours. As the period of usable spot price data is limited and shorter than the workload trace, the spot price trace is looped by the simulator.

4.2 Workload Trace Data

The task workload trace comes from the DAS-2 [11] system and is available from the Grid Workload Archive [8]. The DAS-2 system consists of five clusters totalling 200 nodes with dual 1GHz Pentium III processors. The trace covers the time period from the 22 February 2005 to 13 December 2006. This trace was chosen as it explicitly identifies Bag-of-Task (BoT) applications, which is the case we are considering and allows us to remove other applications from the trace.

In cases where the Requested Time is not specified, the actual run time is used. This does not follow the pattern of the requested time being significantly larger than the actual run time. However this only occurs in 0.14% of the tasks driving the simulation, and as such is not expected to have a significant impact on the results. In our scenario we also consider the Requested Time to be a hard upper bound on the actual runtime of a task and that the middleware enforces this limitation. Hence, the trace data is modified to enforce this limitation by setting the run time of tasks to the requested time as if the task were terminated due to reaching the limit of its requested run time. Almost 6% of tasks have their run time modified in this manner, however on average the runtime is only reduced by less than 1.5 minutes. The trace data used to drive the simulation contains 366 985 tasks with an average execution time of 9.7 minutes per task.

Figure 3 presents a frequency distribution of the per-

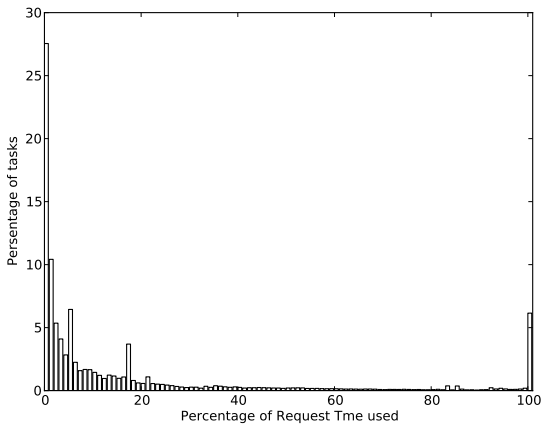


Figure 3. Frequency distribution of the percentage of the requested time that is actually used to execute the tasks.

centage of the requested time that is used to execute each task. The tasks which would otherwise have exceeded the requested time is evident by the spike at 100%. Also 90% of tasks use 67% or less of their requested time to execute and 27.5% of tasks use 1% or less of the requested time. 87.8%[5] of tasks request around 15 minutes of execution time. The large number of tasks using less than one percent of their requested time can be attributed to buggy scripts which terminate after only a few seconds. However, we do not feel that this characteristic invalidates the use of this workload as it is real data that the system we are examining should be able to handle. Also having many tasks that have such a short run time does not aid the system in predicting when deadlines might be breached.

5 Performance Analysis - Results

First we consider at the characteristics of the *Base* and *Base Hard* algorithms and how they are affected by the simulation parameters. We then investigate the available savings, which are possible through the use of spot instances.

5.1 Characteristics of the Base Algorithm

According to Equation 1, having a smaller *Target Ratio* means that the deadline is tighter because the *Maximum Queue Time* is smaller. In Figures 4 and 5 a number of *Target Ratios* are compared.

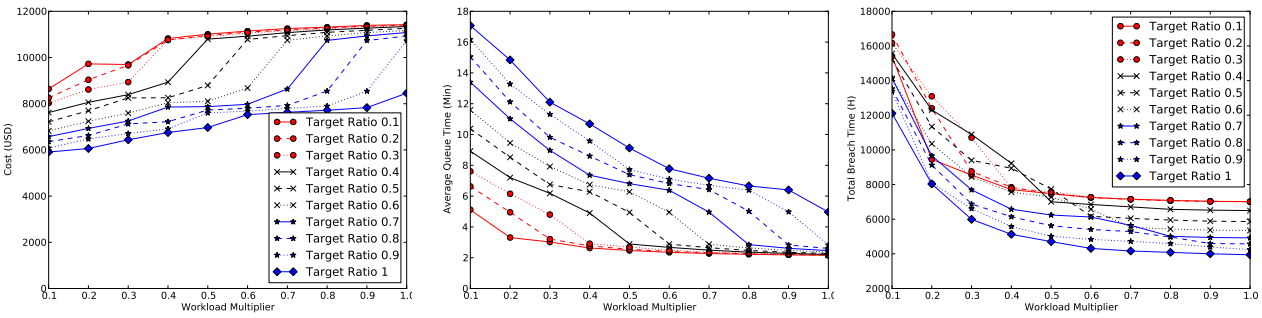
Looking at the graphs for the average queue time 4(b) and 5(b) we see that, as one would expect, the average queue times are lower when the *Target Ratio* is smaller and the deadline becomes tighter. Also as the *Workload Multiplier* increases more and more of the lines converge until at *Workload Multiplier 1*, all but one *Target Ratio* result in a similar average queue time. This indicates that for each *Target Ratio* there is a *Workload Multiplier* above which the tightness of the deadline is no longer a major influence on the overall queue times. In the graphs of the cost, 4(a) and 5(a), this point is even more apparent as for most *Target Ratios* there is a big step from one *Workload Multiplier* to the next.

It is interesting to note that this point occurs immediately after the *Workload Multiplier* reached the same value as the *Target Ratio*. This is the result of large groups of tasks arriving at the same time in the queue with the same *Request Times* and hence having the same deadline by which they should be assigned to a resource. If the *Workload Multiplier* is the same or less than the *Target Ratio* then at least one task can finish on each resource and the next task assigned before the deadline expires. Hence, up to that point in this situation at most half as many resources are needed as tasks in the queue to prevent the predicted breaches of the *Max Queue Time*. When the *Workload Multiplier* is greater than the *Target Ratio* then the expected runtime is also greater than the *Max Queue Time*. To prevent all predicted breaches as many resources are needed as there are tasks in the queue, resulting in many more resources being requested and leading to the jump in cost. Therefore, the algorithm's behaviour has become much more aggressive in requesting resources.

Due to the increased aggressiveness of the algorithms when the *Workload Multiplier* is greater than the *Target Ratio* the number of resources that are requested becomes more heavily influenced by the cap on the number of external resources that can be used at any given time. Hence the *Cost* and *Average Queue Time* of all the *Target Ratios* becomes similar once this threshold is crossed. The fact that this effect is so apparent in the *Cost* graphs (4(a) and 5(a)) highlights the burstful nature of this workload, which is inline with expectations for BoT applications.

In Graph 4(c), one would expect that when the deadlines are more relaxed tasks would end up spending less time in breach. This is the case for higher *Target Ratios*, see the lines for ratios 0.7 and greater which are mostly in order. However with smaller *Workload Multipliers* and *Target Ratios* this order is inverted at times. For example having a *Target Ratio* of 0.1 which is the tightest deadline we trialled, results in a significantly smaller *Total Breach Time* than even a *Target Ratio* of 0.5 when the *Workload Multiplier* is between 0.2 and 0.4. This can also be attributed to the chance in aggressiveness when the *Workload Multiplier* becomes bigger than the *Target Ratio*, as having a *Target Ratio* of 0.1 means that the algorithm has already become more aggressive while with the higher *Target Ratios* are not

[5] 72.9% requested 15minutes and 14.8% requested 16minutes.



(a) Cost (b) Average Queue Time (c) Total Breach Time

Figure 4. *Base* **algorithm with different Target Ratios**

as aggressive with low *Workload Multipliers*.

The *Base Hard* algorithm on the other hand does not rely solely on predicting all breaches; instead it can *react* when a breach is about to occur. Thus the *Total Breach Times* in graph 5(c) are much flatter. As with the *Base* algorithm at *Workload Multiplier 1*, the *Total Breach Time* separates out into distinct levels, only slightly affected by the *Workload Multiplier*. This is the result of the cap on the number of concurrent resources that can be in use at any one time. Without this cap, the *Base Hard* algorithm would be able to request enough resources to prevent all breaches.

5.2 Savings With Spot Resources

To explore the possible savings, which can result from using spot resources we considered three maximum bids as described in section 4.1. Furthermore, a *Target Ratio* of 0.5 will be used for the rest of this analysis as with this ratio the algorithms change to become more aggressive with a *Workload Multiplier* of 0.6 so the behaviour before and after the change should be evident.

The simplest approach to using spot resources is to request them when available without changing the rest of the algorithm. Figure 6 shows the results of using spot resources. The greatest saving can be made when the bid price is the highest, as a high bid price permits greater use of spot resources which cost less per hour than the retail price. At the *Top* bid price no retail resources were used and the *High* bid price, which sits just above the day-to-day spot price fluctuations, resulted in almost the same costs. At the *Low* bid price the system is forced to pay retail price for many resources. The performance in terms of the *Total Breach Time* was essentially unaffected by the use of spot resources at any of the three bid prices. Although we expected the increased volatility at the *Low* bid price to have an effect, at most 33 out of the 366,739 tasks had to be restarted, which can be attributed to the characteristics of the workload.

The *Spot Aggressive* and *Spot Only Hard* algorithms change their behaviour when spot resources are available to request more resources to improve the queue time while only spending a small extra amount. At the *Low* bid price improvement in the *Total Breach Time* is apparent (Graph 6(a)) with minor additional cost seen in Graph 6(b). When using the *High* bid price the *Spot Aggressive* and *Spot Only Hard* algorithms behave much more like the *Base Hard* algorithm, performing slightly worse than it however. This is because at the *High* bid price spot resources are still unavailable at times.

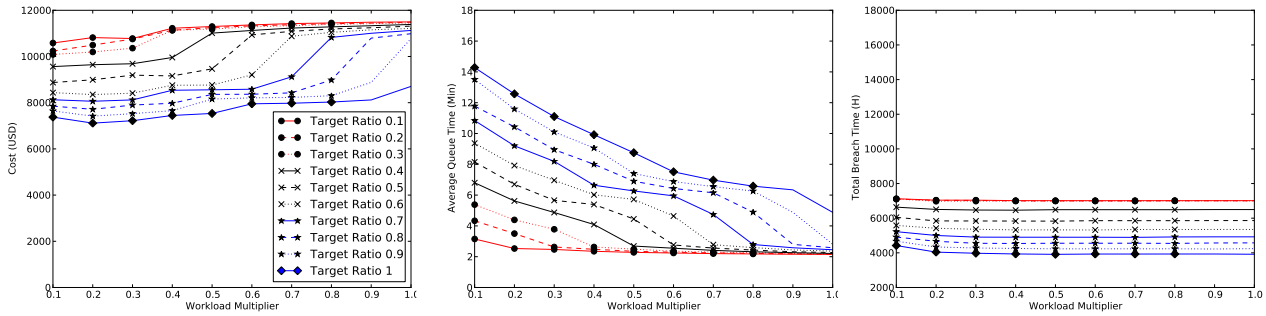
Finally we consider the *Pure Spot* algorithm, which exclusively uses spot resources when available. With a *Low* bit price the performance is heavily influenced by the restricted availability of additional resources(see Table 2). When a higher bid price is used these restrictions are reduced and the performance of this algorithm begins to resemble that of the *Spot Base* algorithm.

5.3 Overall Comparison and Observations

From our analyses so far the following overall observations can be made about using spot resources:

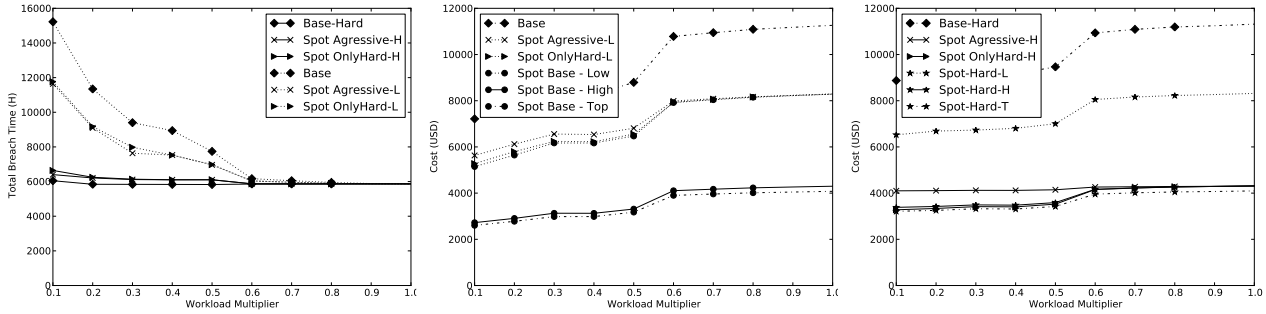
- That there exists a band in which the Spot Price fluctuate most of the time. Hence, even small changes in the bid price to be above this band can have a dramatic impact. In our case a change of USD$0.005 from the *Low* to *High* bid price level reduced the costs by at lead 20%.
- A larger bid price will lower cost, as more work is performed in spot resources rather than on regular full price resources.

For a final concrete comparison a summary of results is presented in Table 2. We choose to only include results with a *Top* bid price of USD$0.13 as they produce the lowest cost, except for the *Pure Spot* policy where the *Low* bid



(a) Cost (b) Average Queue Time (c) Total Breach Time

Figure 5. *Base Hard* **algorithm with different Target Ratios**



(a) Total Breach Time (b) Cost (c) Cost - Hard

Figure 6. Algorithms using spot resources with a Target Ratio of 0.5.

price has the lowest cost. Also the *Spot Aggressive* and the *Spot Only Hard* policies were omitted from the table as at the *Top* bid price they produce the same results as policies presented in the table. The *Target Ratio* used in the table is again 0.5 and results for both a low and a high *Workload Multiplier* are included.

First of all comparing the *Local Only* case of not using any external resources to the performance of the *Base Hard* policy at either *Workload Multiplier* highlights the substantial improvements that are possible. The *Total Breach Time* is reduced to 0.12% of the *Local Only* value, a change of three orders of magnitude. The *Average Queue Time* is taken from over 13 hours down to under 10 minutes and under three minutes with a *Workload Multiplier* of 1.0. This is a reduction of two orders of magnitude.

The amount of money that needs to be spent hiring resources to gain these improvements using the regular fixed pricing model is $8 991.13 or $11 314.69 with a large *Workload Multiplier*. Spread over the 21 months of the workload trace this is around $500 per month. When the flexible Spot Pricing model is used these costs can be lowered to 36% of the retail costs while maintaining the same performance. On a per month basis this is under the $200 mark. Although it is possible to lower these costs further by using the *Pure Spot* approach with a *Low* bid price, the performance, both in terms of the *Total Breach Time* and the *Average Queue Time*, deteriorates so much as not to be worthwhile.

The best approach to take based on our observations is to use a *Spot Base Hard* policy with a very high bid price (close to retail) and choose a *Target Ratio* based on the user's expectations with a *Workload Multiplier* adjusted depending on the importance of the average queue time.

6 Conclusion and Future Directions

In this paper we have discussed some resource provisioning policies that can be used to extend the capacity of a local cluster by leveraging external resource providers, as well as reduce the cost by using the Spot Market.

We introduced two policies that use the amount of requested time for each task to determine the load on the system and when additional resources should be used. We analysed their behaviour from the perspective of the *Workload Multiplier*, which is used to estimate the actual run

	Bid	Workload Multiplier	Total Breach Time (H)	Avg. Queue Time (min)	Cost (USD)
Only Local	N/A	N/A	4 936 934.16	827.44	0.0
Pure Spot Resources	0.06(L)	0.2	1 239 974.83	213.92	1 365.08
Pure Spot Resources	0.06(L)	1.0	1 218 119.96	207.13	1 881.93
Base	N/A	0.2	11 342.33	8.52	7 696.92
Base Hard	N/A	0.2	5 843.20	6.70	8 991.13
Spot Base	0.13(T)	0.2	11 342.33	8.52	2 778.77
Spot Base Hard	0.13(T)	0.2	5 843.20	6.70	3 252.81
Base	N/A	1.0	5 863.08	2.28	11 264.03
Base Hard	N/A	1.0	5 862.98	2.27	11 314.69
Spot Base	0.13(T)	1.0	5 863.08	2.28	4 078.03
Spot Base Hard	0.13(T)	1.0	5 862.98	2.27	4,095.86

Table 2. Summary table of select results with a *Target Ratio* of 0.5

time from the requested time of each task as well as the *Target Ratio*, which determines the 'reasonable' time that tasks can spend in the queue. The experiments using the DAS-2 workload and a small local cluster show that the *Total Breach Time* can be reduced by three orders of magnitude and the *Average Queue Time* by two orders of magnitude while spending less that USD$10 000 over the close to two years that are covered by the trace. Next we explored the savings possible by using spot resources (in conjunction with normal resources) and the impact of choosing a maximum bid price. Concluding that the best saving is achieved when the bid price is set very high, close to the retail price, such that spot instances have a very high likelihood of being used. In that case it is possible to save more than half of the the previous cost while maintaining the same performance characteristics.

For future work we intend to extend the simulations to include other workloads with longer running tasks and include the network overheads involved when assigning a task to a remote resource. The policies will be extended to be aware of applications made up of many individual tasks.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[2] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 197–210, San Francisco, USA, March 2004. USENIX Association.

[3] J. Broberg, S. Venugopal, and R. Buyya. Market-oriented Grids and Utility Computing: The State-of-the-Art and Future Directions. *Journal of Grid Computing*, 6(3):255–276, 2008.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[5] M. D. de Assuncao, A. di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 141–150, New York, NY, USA, 2009. ACM.

[6] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the Cloud: The montage example. In *2008 ACM/IEEE Conference on Supercomputing (SC 2008)*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[7] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny. Inter-operating Grids through delegated matchmaking. In *2007 ACM/IEEE Conference on Supercomputing (SC 2007)*, pages 1–12, New York, USA, November 2007. ACM Press.

[8] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Gener. Comput. Syst.*, 24(7):672–686, 2008.

[9] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for science Grids: a viable solution? In *International Workshop on Data-aware Distributed Computing (DADC'08) in conjunction with HPDC 2008*, pages 55–64, New York, NY, USA, 2008. ACM.

[10] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica. Load balancing in dynamic structured peer-to-peer systems. *Performance Evaluation*, 63(3):217–240, 2006.

[11] The Distributed ASCI Supercomputer 2 (DAS-2). http://www.cs.vu.nl/das2/. 2010-04-28.

[12] Y.-T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[13] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational Grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, Fall 2001.