Exploiting Redundancy for Cost-Effective, Time-Constrained Execution of HPC Applications on Amazon EC2

Aniruddha Marathe
Rachel Harris
David K. Lowenthal
Dept. of Computer Science
The University of Arizona

Bronis R. de Supinski
Barry Rountree
Martin Schulz
Lawrence Livermore
National Laboratory

ABSTRACT

The use of clouds to execute high-performance computing (HPC) applications has greatly increased recently. Clouds provide several potential advantages over traditional supercomputers and in-house clusters. The most popular cloud is currently Amazon EC2, which provides a fixed-cost option (called *on-demand*) and a variable-cost, auction-based option (called the *spot market*). The spot market trades lower cost for potential interruptions that necessitate checkpointing; if the market price exceeds the bid price, a node is taken away from the user without warning.

We explore techniques to maximize performance per dollar given a time constraint within which an application must complete. Specifically, we design and implement multiple techniques to reduce expected cost by exploiting redundancy in the EC2 spot market. We then design an adaptive algorithm that selects a scheduling algorithm and determines the bid price. We show that our adaptive algorithm executes programs up to *7x* cheaper than using the on-demand market and up to 44% cheaper than the best non-redundant, spot-market algorithm.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Cost of Resource Provisioning

Keywords

Cloud; Cost; Resource Provisioning; Fault-tolerance

1. INTRODUCTION

Traditionally, high-performance computing (HPC) users execute scientific applications on dedicated HPC clusters hosted by national laboratories, companies or universities, typically managed through some kind of block allocation or grant mechanism. However, recently the use of cloud resources to execute HPC applications is becoming a popular alternative, due to factors such as machine availability and lower wait queue time. Success stories of scientific applications at HPC scale on the cloud have appeared in the popular press [13]. Unlike standard HPC clusters, however, cloud resources come with variable usage costs for individual users. Cloud resource providers, such as Amazon EC2, offer several *pay-as-you-go* offerings for purchasing cloud resources, which presents a complex optimization problem: What is the most cost effective strategy to execute a given high-performance computing application?

Often, HPC users simply execute their applications on EC2 in the *on-demand market*, which provides dedicated access to a set of machines for a fixed cost per unit time. However, if the application completes before the deadline by which the user requires the results, a second market, the *EC2 auction ("spot") market*, can result in lower cost. While the spot market can provide resources at low cost, jobs are terminated immediately if the current spot price exceeds the bid price. Thus, applications must checkpoint periodically to use the resources productively. Overall, the spot market requires two key decisions: (1) how much to bid; and (2) when to checkpoint.

We explore algorithms to determine the bid price and when to schedule checkpoints for HPC applications that execute on EC2. The algorithms attempt to minimize total user cost while honoring a user-specified application time bound. In one of our key contributions, our algorithms exploit redundancy across multiple groups of EC2 resources, so-called *zones*, to obtain higher availability. We show that, despite higher up-front cost, redundancy often results in lower total application cost because of less frequent downtimes and therefore lower checkpoint frequency.

Each algorithm has its strengths and weaknesses, which leads us to an adaptive algorithm that automatically selects from the algorithms based on current conditions. Our adaptive algorithm uses past spot price behavior to determine an effective algorithm along with an effective bid price. We also consider a relatively simple but often effective scheme that simply bids an excessively large amount, which avoids termination at the risk of higher cost.

We make the following contributions in the paper.

- We show that on the EC2 spot market, checkpoint-

HPDC'14, June 23–27, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-2749-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600212.2600226 .

insertion algorithms using redundancy typically result in lower cost than their non-redundant counterparts.

- We analyze and categorize situations in which the different algorithms perform well.
- We develop an adaptive approach that automatically selects an algorithm based only on past spot price behavior.

Our evaluation revealed several insights. Compared to the naive approach of using on-demand, our adaptive scheme yields up to *7x* lower cost. In addition, our adaptive scheme executes programs up to 44% cheaper than the best-case existing non-redundant algorithms that use the spot market. In comparison to an approach in which a user simply bids a large amount in order to avoid job termination, our adaptive scheme provides a significant advantage in avoiding situations in which the cost is much larger than simply using the on-demand market.

The paper is organized as follows: Section 2 provides the necessary background, and Section 3 describes how we exploit redundancy. We describe our algorithms in Section 4, the experimental setup in Section 5, and the experimental results in Section 6. These results motivate the need for an adaptive policy, which is described and evaluated in Section 7. We then describe related work in Section 8 and conclude in Section 9.

2. OVERVIEW

In this section we first describe the mechanics of the EC2 spot market and define the problem of selecting a fault-tolerance mechanism for time-constrained runs. Next, we describe why our work on the spot market on Amazon EC2 is relevant, in general, to clouds. Finally, we describe our system model.

2.1 EC2 Spot Market

The standard offering from Amazon EC2, known as *on-demand* pricing, guarantees resource availability for an hour of use at a fixed rate. At the end of every hour, the contract between the user and the cloud provider is renewed, and resource usage is granted for the next hour. Alternatively, EC2 auctions unused resources, which Amazon denotes the *spot market.* Spot prices can be significantly less than their on-demand counterparts for high-end EC2 resources. Popular HPC offerings such as Cluster Compute Eight Extra Large (CC2) instances are as much as eight times less expensive on the spot market as their corresponding on-demand prices. On the spot market, the user selects a *bid price*, and EC2 grants the resource if the bid is higher than the (EC2-maintained and demand-based) spot price. However, the system terminates the resource *immediately* and without warning if the spot price moves above the bid.

The spot market employs the following set of rules:

- Hour-boundary pricing: The user is charged for the hour based on the spot price (*not the bid price*) at the start of the hour. Spot price movements within the user's bid price do not affect the rate for that hour.
- Partial-hour usage: Partial-hour resource usage due to abrupt termination by EC2 is not charged to the user.
- Fixed bid: Once a spot request is submitted, the user cannot alter the bid. To change the bid, the user must cancel the spot request and submit a new request.
- Abrupt termination: EC2 does not notify the user before terminating a resource.
- Uncertain wait time: The user does not acquire a resource when the spot price is larger than the bid price.

To exploit low spot market prices for running tightly coupled HPC applications, one must use fault-tolerance techniques. Generally, HPC applications use checkpointing for such situations, which has a tradeoff between the overhead of checkpoints and how much computation is lost when a failure occurs. Previous work in this area focuses on predicting failures by analyzing real-time spot price data (see Section 8).

Running applications on the spot market does not provide a guaranteed completion time. Many scientific HPC applications must complete within a user-defined time bound in order to be useful. The bound depends upon context (e.g., "finish the weather prediction for tomorrow before the evening newscast at 7pm"). Typically, the deadline is further from the current time than the application takes to complete assuming uninterrupted execution. The difference between the deadline and the earliest possible completion time is *slack.* Given non-zero slack, the spot market can be used. However, the application then requires an algorithm to schedule checkpoints that minimizes the total cost.

2.2 Relevance to General Clouds

Our research on Amazon EC2 is relevant and applicable to the cloud in general. Amazon EC2 has become a popular platform for running scientific applications cost-effectively in recent years. In the HPC arena, Amazon EC2 has been evolving to provide top-of-the-line, HPC-grade compute resources securing a high ranking in the well-known Top-500 supercomputer list [15]. Stories of hero-type runs on EC2 frequently appear in the news [13]. Recent success stories on running scientific applications on the spot market present an attractive performance-per-dollar trade-off compared to existing, institution-owned HPC clusters [14]. A recent study shows that Amazon EC2 is significantly larger in compute capacity than their competition *combined* [4]. Consequently, the problem of selling unused capacity during sluggish demand is much more important to Amazon than other providers and will continue to exist for current and future cloud providers. Note also that our work, while targeted towards MPI applications in this paper, is in no way dependent on MPI.

Amazon's spot price mechanism is, in our view, not likely to change in the future. The objective behind the way Amazon's spot market has been structured is two-fold. One goal is to attract a sufficient number of users at a significantly discounted price compared to on-demand, so that the operating costs of already running unused instances is recovered. The second goal is to prevent users from monopolizing resources through the spot market, which would decrease Amazon's profit. The discounted nature of spot prices satisfies the first objective. On the other hand, Amazon provides no guarantees on instance up time via abrupt termination along with a fixed bid price, which accomplishes the second objective. Work by Ben-Yehuda et al. [1] on statistically analyzing the

Term	Description
t_c	Checkpoint cost
t_r	Restart cost
B	User's bid price
S	Spot price
C	Total uninterrupted computation time
C_r	Remaining computation time
D	User-defined time bound (deadline)
P	Progress made
T	Current time
T_r	Remaining time
T_u	Instance up-time
T_l	Slack time
T_s	Scheduled checkpoint time

Table 1: System model variables.

spot prices concluded that the spot prices are not purely based on user bid or resource supply, and hence users may not be able to make well-informed predictions of the spot prices over a long period of time.

Modifications to the spot market are full of practical problems. For example, instead of abrupt termination, the user could be provided with a notification beforehand. Another modification could be to notify the user about an out-of-bid situation and charge at a higher rate for a shorter billing cycle (less than one hour), allowing for a checkpoint before termination. Briefly, such schemes generally are undesirable to Amazon, because they could lead to fewer users using the (higher profit-generating) on-demand market. Appendix A discusses in detail the ramifications of such modifications.

2.3 System Model

Next, we present definitions for our underlying system model of both the Amazon EC2 spot market and checkpoint scheduling mechanisms. EC2 offers top-of-the-line HPC compute clusters, labeled *CC2 instances*, with node performance competitive to computing resources found on traditional dedicated HPC clusters at universities or national laboratories. In this paper, based on our previous experience as well as that of other groups [11], we use the spot market to run only CC2 instances and ignore other inferior clusters.

The user specifies an *experiment* as a configuration of a number of nodes, problem size, execution time and job completion deadline. We denote as C the (user-provided) execution time for the given number of nodes and problem size, i.e., the time to execute the application under the EC2 on-demand option with no system interruptions. The user also provides the deadline, which we denote as D, which is the time span in which the job *must* complete ($D \geqslant C$). We denote the *slack* between the deadline and the given execution time as T_l ($T_l = D - C$).

Let B be the user's bid price, which is the maximum amount the user is willing to pay per hour, and let S be the spot price. When S exceeds B, the currently running spot instance is terminated. Similarly, when S becomes less than or equal to B, a currently submitted spot instance is initiated. We assume constant checkpoint and restart costs for a configuration denoted by t_c and t_r respectively. Variable T_s denotes the time at which a checkpoint is initiated. Table 1 summarizes the variables in our model.

Figure 1 provides an example. The x-axis shows the progression of time and the y-axis shows price per hour. Plot

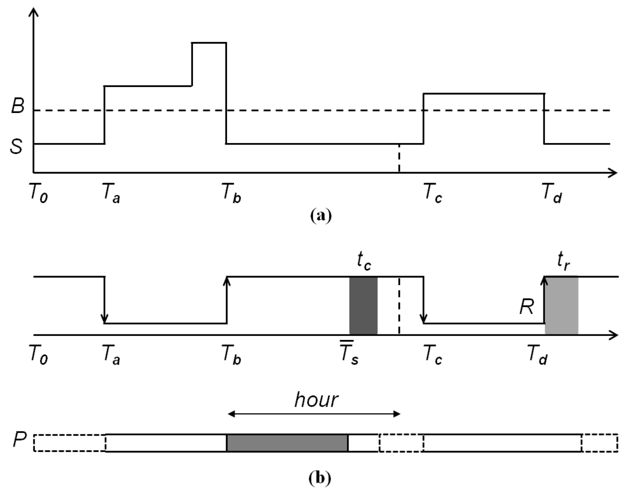

Figure 1: Spot price movements and state transitions: (a) spot price movements and progress; (b) instance states and checkpoint-restart costs.

(a) shows a scenario on the spot market with a user bid of B and movement over time of spot price S. Plot (b) shows the state transitions of the instance corresponding to the spot price movements relative to B. The instance starts at T_0 because $S < B$. At time T_a, $S > B$, so the instance is terminated. When S is again less than B at time T_b, the instance is re-initiated. The application restarts from its initial state since no checkpoint was taken (so its state was lost at T_a). The user schedules a checkpoint at $\overline{T_s}$ where the system takes t_c time to checkpoint (shown in dark grey). Termination again occurs at T_c, as does instance re-initiation at T_d. However, the application restarts from the checkpoint taken at $\overline{T_s}$. The restart operation R takes t_r time (shown in light grey). The user is charged the value of S at T_b for the first hour. Net application progress, denoted by P, is shown by the grey horizontal bar at the bottom of plot (b). Dotted boxes denote speculative progress that is not committed by a checkpoint. Empty boxes denote no computational progress due to a checkpoint, a restart, or system downtime.

3. EXPLOITING REDUNDANCY

EC2 auctions computational resources at different data centers (known as *availability zones*) at independent bid prices. For applications with significant slack and low checkpoint cost, bidding in a single zone results in low cost while still meeting the deadline. However, for applications with little slack or high checkpoint cost, or during times of spot price volatility, bidding only in a single zone can incur:

- Low system availability while the spot price is high;
- High checkpoint overhead; or,
- High rollback costs.

For such situations, the user could simply increase the bid. However, this choice does not guarantee high system availability at *low cost*, due to the nature of spot price movements (see Section 6). Thus, we introduce redundancy as an alternate, complementary fault-tolerance mechanism.

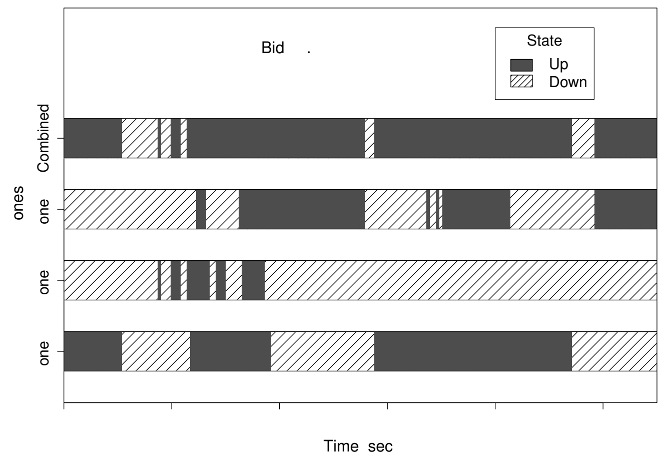

Figure 2: Availability for the three CC2 zones in the US-East region and the combined availability during a 15 hour period on December 19, 2012.

3.1 Redundancy in Independent Zones

Figure 2 shows an example of three CC2 zones in the US-East region. The solid portions show when the zones are up, and the textured portions when they are down. The top bar shows the combined up time; i.e., the times that at least one zone is up. Clearly, redundancy demonstrates potential for significantly increased up time in situations where individual zones have volatility.

However, for redundancy to be a viable solution, the movements in spot prices in different EC2 zones must be sufficiently independent. To investigate the interdependence of prices in different zones, we employed a *Vector Auto-Regression*, using the Akaike criteria to determine the optimal number of lags. As expected, each zone has a strong dependency on its own price history. Though there is some statistical significance in the dependencies across zones, the size of the effect is consistently 1-2 orders of magnitude smaller than within a zone. This difference of magnitude in same-zone vs. across-zone lagged price effects indicates an opportunity for computational arbitrage.

3.2 General Algorithm to Determine Checkpoints with Redundant Zones

In order to use the spot market efficiently across multiple zones, we need new algorithms to determine when to checkpoint. These algorithms lead to a set of policies to optimize use of spot market resources.

We start with the base algorithm that Algorithm 1 shows and extend it in the following section. The base algorithm alone extends prior work in two ways: first, it guarantees completion within the user time bound D; and second, it allows use of multiple zones. Our algorithm takes several parameters as input—the number of zones (degree of redundancy), $N \geqslant 1$; the bid and spot prices (B and S_i); and the checkpoint and restart costs (t_c and t_r) — and determines when to initiate checkpoints.

A zone is considered *up* when a spot instance is requested, and $B \geqslant S$ for the zone. Each zone runs a separate MPI application in its entirety with a fixed number of (user-specified) virtual machine instances. In the algorithm, $Instance_i$ refers to the application executing on zone i ($\forall i \in N$). Based on the conditions on the spot market and the remaining time T_r, the algorithm chooses between on-demand and the spot market and selects N if the spot market is chosen. We assume that the algorithm monitors application progress, P, through an interface; e.g. `MPI_Pcontrol` is often used to indicate iteration completion in iterative MPI applications. Because the algorithm continuously monitors T_r, it can potentially handle changes in the input parameters such as the deadline D (modified by the user during application runtime) or variation in application performance (which affects P).

Algorithm 1 Algorithm framework to schedule checkpoints. Inputs are number of zones, N, bid and spot prices (B and S_i), and checkpoint and restart costs (t_c and t_r). $Instance_i$ is initially down ($\forall i, 1 \leqslant i \leqslant N$).

```
1: while $T_r$ != 0 && $C_r$ != 0 do
2:   for i = 1 to $N$ do
3:     if $Instance_i$ is up and $B < S_i$ then
4:       $Instance_i \leftarrow$ down;
5:     else if $Instance_i$ is down and $B \geqslant S_i$ then
6:       $Instance_i \leftarrow$ waiting;
7:     end if
8:   end for
9:   $T_r \leftarrow D - T$
10:  $C_r \leftarrow C - P$
11:  if ($T_r == C - P + t_c + t_r$) then
12:    /* switch to on-demand to meet deadline */
13:    Checkpoint();
14:    RestartOnDemand(); /* single zone */
15:  else
16:    if ($\exists\ i \in N \mid Instance_i$ is up) then
17:      if ($CheckpointCondition()$) then
18:        $P \leftarrow$ Checkpoint();
19:        for i = 1 to $N$ do
20:          if $Instance_i$ is waiting then
21:            $Instance_i \leftarrow$ up;
22:            RestartFromRecentCheckpoint();
23:            ScheduleNextCheckpoint();
24:          end if
25:        end for
26:      end if
27:    else
28:      /* No zone is up */
29:      for i = 1 to $N$ do
30:        if $Instance_i$ is waiting then
31:          $Instance_i \leftarrow$ up;
32:          RestartFromPreviousCheckpoint();
33:          ScheduleNextCheckpoint();
34:        end if
35:      end for
36:    end if
37:  end if
38:  Compute(); /* run on all zones that are up */
39: end while
```

Lines 2-8 update the state of $Instance_i$ based on B and S_i. Lines 9 and 10 update the current T_r and C_r respectively. Line 11 ensures that the deadline D will be met by using on-demand market if the remaining time is equal to the remaining computation plus migration overhead (i.e, a checkpoint and a restart). If at least one zone is up and $CheckpointCondition()$ is true, then lines 16-18 take

Algorithm 2 *ScheduleNextCheckpoint*() for Markov-Daly policy.

```
1: $ScheduleNextCheckpoint()$
2: {
3:     $E[T_u] \leftarrow$ expected_uptime($B$, $i \in N$);
4:     $T_s \leftarrow T+$ opt_ckpt($E[T_u]$, $t_c$);
5: }
```

a checkpoint, update progress and line 23 schedules a new checkpoint.

To avoid checkpoint and restart overheads every time a zone is re-started, we introduce *waiting* state. Lines 5-6 mark a zone that is eligible to run as waiting ($B \geqslant S_i$; but no spot instance is requested on the zone). Thus, the zone can receive a checkpoint from another zone before starting. A running zone that takes a checkpoint at line 18 restarts waiting zones (lines 19-22) from that checkpoint by requesting a spot instance on the zone and marks them as running. If no zone is running, lines 29-33 restart all waiting zones from a previous checkpoint, the zones are marked as running, and the next checkpoint is scheduled. The algorithm is generic and can accept any *CheckpointCondition*() and *ScheduleNextCheckpoint*(). We define each policy, as described in the next section, by these two functions.

4. REDUNDANCY-BASED POLICIES

In this section we describe our policies that exploit redundancy. In turn, we describe our *Periodic*, *Markov-Daly*, *Edge*, and *Threshold* checkpointing policies.

4.1 Periodic policy — checkpointing at hour boundaries

Given N zones, *ScheduleNextCheckpoint*() (see Algorithm 1) schedules a checkpoint at regular intervals (at the end of every hour in this paper) such that the checkpoint *completes* within the hour boundary ($T_s = hour - t_c$) [18]. The user is charged S at the end of each hour, as long as $B > S$ throughout the hour. Function *CheckpointCondition*() returns *true* when $T = T_s$.

4.2 Markov-Daly policy — predicting up time

Building on previous work [2] to predict up time for single zone cases (and without considering checkpointing), we use a variant of the Chapman-Kolmogorov equation to get the expected up time of a zone using the zone's price history. Algorithm 2 shows the basic idea behind *ScheduleNextCheckpoint*(): first, calculate the expected uptime given a bid price; then, use that expected up time to determine the optimal checkpoint frequency. The Markov model produces the expected up time. We then use Daly's equation [3], a well-known tool to calculate optimal checkpoint frequency to safeguard against hardware failures, to obtain the optimal checkpoint frequency. As with *Periodic*, *CheckpointCondition*() simply checks if $T = T_s$. We calculate $E[T_u]$ for each zone on line 3 of Algorithm 2. For zones with *independent* price movements (see Section 3), the combined $E[T_u]$ is the sum of $E[T_u]$ of individual zones. Thus, $E[T_u]$ for the replication-based scheme is necessarily larger than with individual zones. We combine $E[T_u]$ and t_c as input to Daly's equation to calculate the optimal checkpoint frequency, which decreases as N increases. The Appendix fully details our Markov approach.

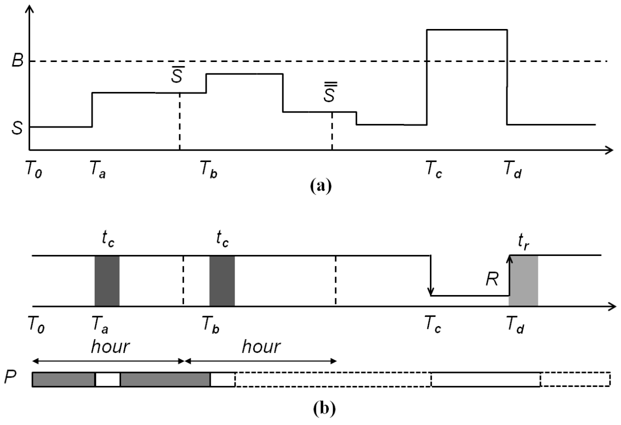

Figure 3: Rising Edge checkpoint policy. Part (a) shows user bid and spot price movements; Part (b) shows instance states, checkpoint-restart events and costs, and net progress.

4.3 Rising Edge policy — reacting to rising price

This algorithm (referred to as *Edge* hereafter) sets *CheckpointCondition*() from Algorithm 1 to *true* whenever an upward movement occurs in the spot price S in an executing zone [18]. The upward movement indicates that $S > B$ might occur soon. Hence by taking a checkpoint, progress is saved. Function *ScheduleNextCheckpoint*() is a no-op, because the checkpoint decision is made instantaneously based solely on current values of B and S. Figure 3 shows the steps involved in the *Edge* policy. For a zone with relatively stable spot prices, the *Edge* policy saves checkpoint costs compared to periodic checkpointing, but can lose substantial progress if the spot price increases sharply.

4.4 Threshold checkpoint scheduling policy — reducing cost of Edge policy

Previous work on scheduling checkpoints in a single zone [7] describes an algorithm that is an outgrowth with the *Edge* policy. The algorithm operates on two thresholds. *CheckpointCondition*() is set to true if either of the following two conditions is true in an executing zone. First, a price threshold *PriceThresh* is calculated as the average of minimum spot price S_{min} and B. The first condition is *true* when S shows a rising edge and $PriceThresh \leqslant S$. Second, a time threshold *TimeThesh* is calculated as the probabilistic average up time of a zone. Another variable, execution time at B, equals the up time at bid price B since the most recent restart or checkpoint. The second condition is *true* when *TimeThresh* is less than execution time at B. *ScheduleNextCheckpoint*() schedules an immediate checkpoint if either condition is true.

5. SIMULATION SETUP

This section describes our assumptions about the working environment, applications and checkpointing policies. We make the following assumptions about experiment configurations to evaluate policies presented in the paper.

- The problem size and number of MPI tasks are fixed for an experiment.
- Bid prices for all nodes in a zone are identical.
- Checkpoints are stored onto an I/O server that runs in an on-demand instance as long as spot instances are running.

HPC users typically run MPI applications on the spot market along with one or more I/O instances through the on-demand option with persistent storage (e.g., EBS volumes on EC2) attached [9]. A typical I/O server setup (non-CC2) at the on-demand price costs only a fraction of the total cost of running a tightly coupled MPI application at scale. Hence, we ignore the cost of running such I/O server setup in our experiments. A sophisticated multi-I/O server setup has been discussed elsewhere and is beyond the scope of this work [9].

Recovery costs for an MPI application on the spot market include instance queuing delay, which is the time between the submission of the spot request ($S \leqslant B$) to the time when the instance is accessible. Previous studies have shown that Amazon EC2 instances incur a measurable boot time on the on-demand option (order of several minutes in the worst-case) [11]. We measured the queuing delay on the spot market for CC2 instances by submitting spot instance requests at 7:00 AM and 7:00 PM every day for two months with the bid price equal to the instantaneous spot price. We measured the time between the submission of the spot request to the time when the instance was available for login by attempting to establish an SSH connection to the instance every ten seconds after the instance became "*running*". We observed an average queuing delay of 299.6 seconds with best-case and worst-case delays of 143 seconds and 880 seconds, respectively. The queuing delay on the spot market contributes to an added penalty to application recovery cost.

Another major factor contributing to checkpoint costs originates from inferior network bandwidth on the cloud. We confirm previous findings that showed that when using system-level checkpointing, MPI benchmarks from the NAS benchmark suite showed up to 200 seconds of overhead for small problem sizes at up to 64 tasks [5]. Due to limited funds, we could not perform extensive runs on EC2 to measure checkpoint costs for real applications with large working sets. Previous work shows that real applications spend a significant portion of the hour in checkpoint and restart operations (up to tens of minutes) [9] and we therefore assume checkpoint and restart overheads compatible with the range of existing studies of 300 to 900 seconds. For simplicity, we also assume checkpoint and restart costs are equal. Although resource acquisition on the spot market involves variable delay even in perfect conditions ($S \leqslant B$), our assumptions about fixed restart costs do not negatively affect the correctness of our work.

For the purpose of simulation, we assume an uninterrupted application execution time of 20 hours. We choose slack values of 15% to 50% (3 to 10 hours). We use the spot price history of CC2 instances with Linux of over 12 months (between December 2012 to January 2014). The state of spot prices in all zones is sampled at a 5-minute interval for all three zones. Spot price movements within a 5-minute interval (although present) are rare and hence the loss in precision does not affect the key findings in our work. We observe several low, moderate and high spot price volatility windows in our 12-month data. However, for representative results, we use low spot price volatility and high spot price volatility windows for evaluation of our policies over different spot price behaviors. For low spot price volatility window, we use the spot price data for March 2013 with average spot price of \$0.30 and a variance of less than 0.01 in each zone. Similarly, for the high spot price volatility window, we use the spot price data for January 2013 with average spot prices between \$0.70 to \$1.12 and a variance of up to 2.02 in each zone. We run 80 experiments over partially overlapping chunks in each spot price window. To build the system state for *Markov-Daly* policies, we use a price history size of 2 days and we assume bid prices between \$0.27 to \$3.07 in steps of \$0.20. We use bid prices larger than \$2.40 to avoid failures due to occasional spot price spikes of up to \$3.00.

6. EVALUATION OF POLICIES

We now show the effectiveness of the various policies discussed in Section 4. Figure 4 compares different single-zone checkpointing policies with the *best-case* redundancy-based policy. The comparison is shown for low and high volatility windows as well as low and high slack values; in addition, the checkpoint cost is fixed at 300 seconds. For each single-zone checkpoint policy, we merge the results from all three individual zones (each of which could be selected by a user) to generate one boxplot. Similarly, due to space limitations and consistency, we pick the best-case redundancy-based policy for each experiment, though our redundancy-based policies perform fairly similarly (with *Markov-Daly* performing slightly better than others). We observed diminishing returns with $N \leqslant 2$ zones for redundancy, and so we do not include this case in our evaluation. The x-axis shows checkpoint scheduling policies, and the y-axis shows the total cost per instance in dollars. The boxplot shows the variation in cost of execution at different slack values (denoted T_l) for low (top) and high (bottom) volatility windows. For plots (a) and (c), T_l = 15% of computation time, and for plots (b) and (d), T_l = 50%. Due to limited space, we do not show the boxplots for high checkpoint cost (900 seconds). We summarize which policies have lowest cost in Table 2 (low checkpoint costs) and Table 3 (high checkpoint costs).

We make several observations from the boxplots. First, at low T_l (plots (a) and (c)), spot price volatility determines if single-zone or redundancy-based policy results in the least cost. In general, low volatility results in higher availability (uptime) per zone at low bid. For time-constrained execution, higher availability on the spot market results in lower overall costs due to lower use of the (expensive) on-demand option. In the case of low volatility (plot (a)), *Periodic* is superior due to low checkpoint cost (which in our implementation of *Periodic* is incurred hourly) as well as infrequent restarts (because with low volatility, there are fewer failures). This is true even at low bids. On the other hand, for high volatility, redundancy generally results in lower cost (plot (c)). This is because of (1) lower checkpoint overhead and (2) higher combined system availability at a lower bid. For example, in the case of low T_l with high volatility (plot (c)), the best-case redundancy-based policy results in 23.9% lower costs than *Periodic*, which is the best-case existing single-zone policy in this case.

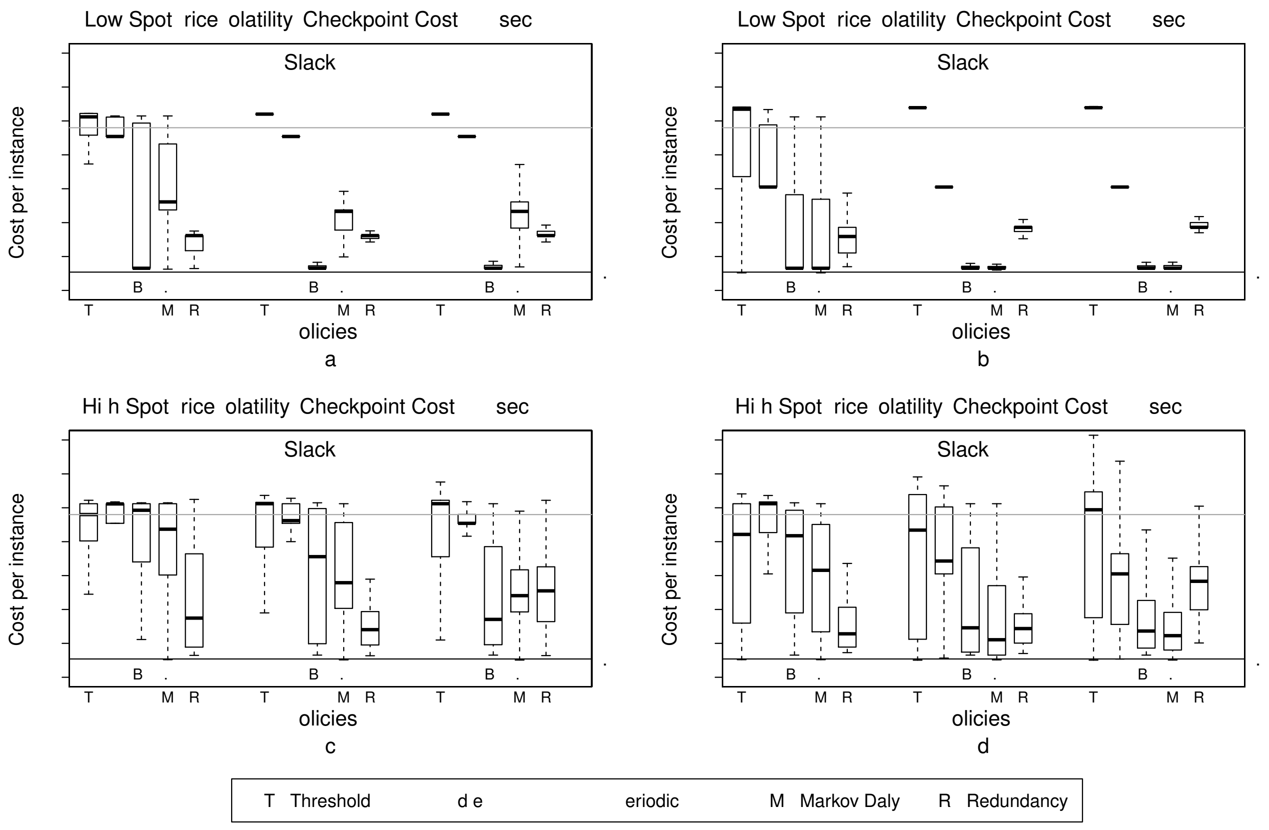

Figure 4: Comparison of single-zone checkpoint and best-case redundancy-based policies for checkpoint cost = 300 seconds at B = \$0.27, \$0.81 and \$2.40. The comparison is shown for two data-sets: low volatility (top row) and high volatility (bottom row). A horizontal grey line at \$48.00 shows the cost at on-demand rate (\$2.40/hr), and a horizontal black line at \$5.40 shows reference cost at lowest spot price (\$0.27/hr). Protocol abbreviations are <u>T</u>hreshold, Rising <u>E</u>dge, <u>P</u>eriodic, Single-Zone <u>M</u>arkov-Daly, and <u>R</u>edundancy-based (best-case)

Second, at a high T_l (plots (b) and (d)), single-zone policies generally show lower costs than redundancy-based policies because a single zone has a sufficiently high probability of executing solely on the spot market at a low bid price. Again, low or high volatility influences the cost difference between single-zone and redundancy-based policies. In case of low volatility (plot (b)), *Periodic* and single-zone *Markov-Daly* show lower median costs (confirming prior results for *Periodic* [18]). For high volatility (plot (d)), the best-case redundancy-based policy shows median costs similar to single-zone policies. In this case, median costs depend on particular price movements in individual experiments.

Third, redundancy generally shows better median costs at lower bid prices ($B \leqslant \$0.81$) due to the lower possibility of paying for all three zones at a higher combined availability. Higher T_l results in lower worst-case costs but does not significantly affect the median costs of redundancy-based policies. This is because the checkpoint/restart overhead is already low and system availability is already high, so there is little added benefit from additional slack. As mentioned earlier, Table 2 summarizes the results for low checkpoint costs.

Table 3 summarizes the best policies for configurations with large checkpoint costs (900 seconds). For low and high volatility windows with low T_l, the best-case redundancy-based policy yields the lowest median-costs. (up to 56% better than the best single-zone policy). For high T_l, low spot price volatility results in single-zone *Periodic* as well as *Markov-Daly* yielding the lowest median costs; this is due to their low checkpoint frequency when slack is high. For high spot price volatility and high T_l, *Markov-Daly* yields the lowest median costs. We observe that *Edge* and *Threshold* policies result in high median costs due to high recovery costs resulting from inadequate checkpointing at low bid prices. This confirms prior results [8]. Hence, we exclude *Edge* and *Threshold* in all further evaluation.

Spot price volatility	Slack	
	15%	50%
Low	Periodic (bid = \$0.81)	Periodic/Markov-Daly (bid = \$0.81)
High	Redundancy (bid = \$0.81)	Markov-Daly (bid = \$0.81)

Table 2: Optimal policies for the experiments with $t_c = 300$ seconds, low (15%) and high (50%) slack for low and high spot price volatility windows.

Spot price volatility	Slack	
	15%	50%
Low	Redundancy (Bid = \$0.27)	Periodic/Markov-Daly (Bid = \$0.81)
High	Redundancy (Bid = \$0.81)	Markov-Daly (Bid = \$2.40)

Table 3: Optimal policies for the experiments with $t_c = 900$ seconds, low (15%) and high (50%) slack and low and high spot price volatility windows.

Summary.

The critical point here is that for different experiment configurations and bid prices, different protocols result in the best costs. Spot price volatility also influences the median costs. In addition, the boxplots above show only a small subset of the permutations of bid price, checkpoint policy and number of zones from which the user can select. Note that in general single-zone *Periodic* shows better median costs at $B = \$0.81$, whereas, higher bid prices result in better costs for single-zone *Markov-Daly*. Higher bid prices (after a sweet-spot) generally increase the median cost for redundancy-based policies as a result of paying more for additional zones. These factors motivate an adaptive mechanism to select the most appropriate policy given past and current conditions on the spot market. In the next section, we design, implement, and evaluate such a mechanism.

7. ADAPTIVE POLICY

As we showed in Section 6, the best choice of policy changes depending on the condition of the spot market. This section first explains our design and implementation of an "adaptive policy" (denoted *Adaptive* hereafter) that can switch checkpoint or redundancy-based policies dynamically depending on spot market conditions. Next, we evaluate *Adaptive*, which shows that we achieve two broad results. The first is that *Adaptive* typically results in a policy that is as good or nearly as good as the best policy from Section 4. The second is that *Adaptive* in general *avoids* choosing a policy that leads to high cost.

7.1 Description

The optimal algorithm for time-constrained execution of an experiment depends on two fixed parameters (the slack (T_l) and the checkpoint cost (t_c)), and three variables (bid price (B), number of zones used (N), and the policy used). It also depends on changes in the spot price (S). From a user perspective, the problem of selecting the correct policy to optimize cost while completing in the time-bound is nontrivial. Furthermore, the best policy changes over time depending on S. Finally, the values of B and N need to be chosen; the algorithms in the previous subsection do not indicate how to choose them. In this section we describe our novel *Adaptive* checkpoint scheduling scheme, which chooses an effective policy as well as B and N.

Adaptive works as follows. First, it boot-straps by reading the spot price history prior to the experiment start time to load the "current state". At each 5-minute step, *Adaptive* simulates cost and computation for each permutation of B, N, and policy; B is chosen in a range of \$0.27 to \$3.07 (the upper bound covers occasional spikes) in steps of \$0.20 and N is 1, 2, or 3. During an experiment, *Adaptive* selects a new permutation, if any of the following is true: (1) the current zone has been terminated due to $S > B$; (2) the billing hour has ended; or (3) the new policy does not change the running zone or B in the current billing hour.

A new configuration is chosen at run-time as follows. To guarantee completion by the user's deadline, *Adaptive* considers the following inequality:

$$C_r - T_r \times \frac{P}{T} > 0 \qquad (1)$$

where, C_r is the remaining computation, T_r is the remaining time to meet the deadline, and $\frac{P}{T}$ denotes the current rate of progress (see Table 1). *Adaptive* evaluates $\frac{P}{T}$ for each permutation of B, N, and policy. For a permutation, if the left hand side of the inequality is positive, then a switch to on-demand will occur; otherwise, only spot market will be used at the current rate of progress (assuming for the moment that there is no cost to checkpoint and restart during the switch to on-demand). To select the least-cost policy, *Adaptive* estimates, for each permutation, (1) the time on the spot market before the switch to on-demand and (2) the remaining time on on-demand (using Inequality (1)). It is straightforward to estimate the total remaining cost based on the current rate of expenditure on the spot market and the fixed rate of \$2.40 for on-demand. Then, *Adaptive* chooses the permutation with the least predicted remaining cost. To add checkpoint and restart costs, we merely place their sum as a term on the left-hand side of Inequality (1).

7.2 Evaluation

We show the effectiveness of *Adaptive* with two experiments. First, we evaluate the cost-effectiveness of *Adaptive* against existing protocols. Second, we compare *Adaptive* to a simple but often effective policy called *Large-bid* [8].

7.2.1 Comparison to Single-Zone and Redundancy-Based Protocols

Figure 5 compares *Adaptive* with best-case single-zone checkpointing policies (*Periodic* and *Markov-Daly*) as well as the best-case redundancy-based policy. As in the previous subsection, we merge the boxplots of single-zone checkpoint policies from each zone for a fair comparison. We observe that $B = \$0.81$ generally results in better median costs compared to other bid prices. Hence, we choose the bid price of \$0.81 to compare best-case costs of policies (see Figure 4).

The key point is that *Adaptive* is always at least competitive with the *best* of the other three algorithms, and which is best (and worst) changes with spot price and slack. Therefore, *Adaptive* avoids situations that would lead to large costs to the user. In the case of low volatility (plots (a), (b), (c) and (d)), *Adaptive* quickly converges to the best-case single-zone or redundancy-based policy for different T_l and t_c values, resulting in similar median as well as worst-case costs. Specifically, low T_l values (plots (a) and (b)) result in median costs of *Adaptive* that are comparable to existing policies, but show a smaller range of second and third quartile costs (and so have low variance). Higher checkpoint costs ($t_c = 900$ sec) (plot (b)) result in median cost for *Adaptive* that is up to 44.2% lower than the best-case median cost across all bid prices for the existing single-zone policy (which, in this case, is *Periodic*). Note that the comparison is between *Adaptive* and the *best-case* existing policy—and a user will not in general be able to determine the best-case

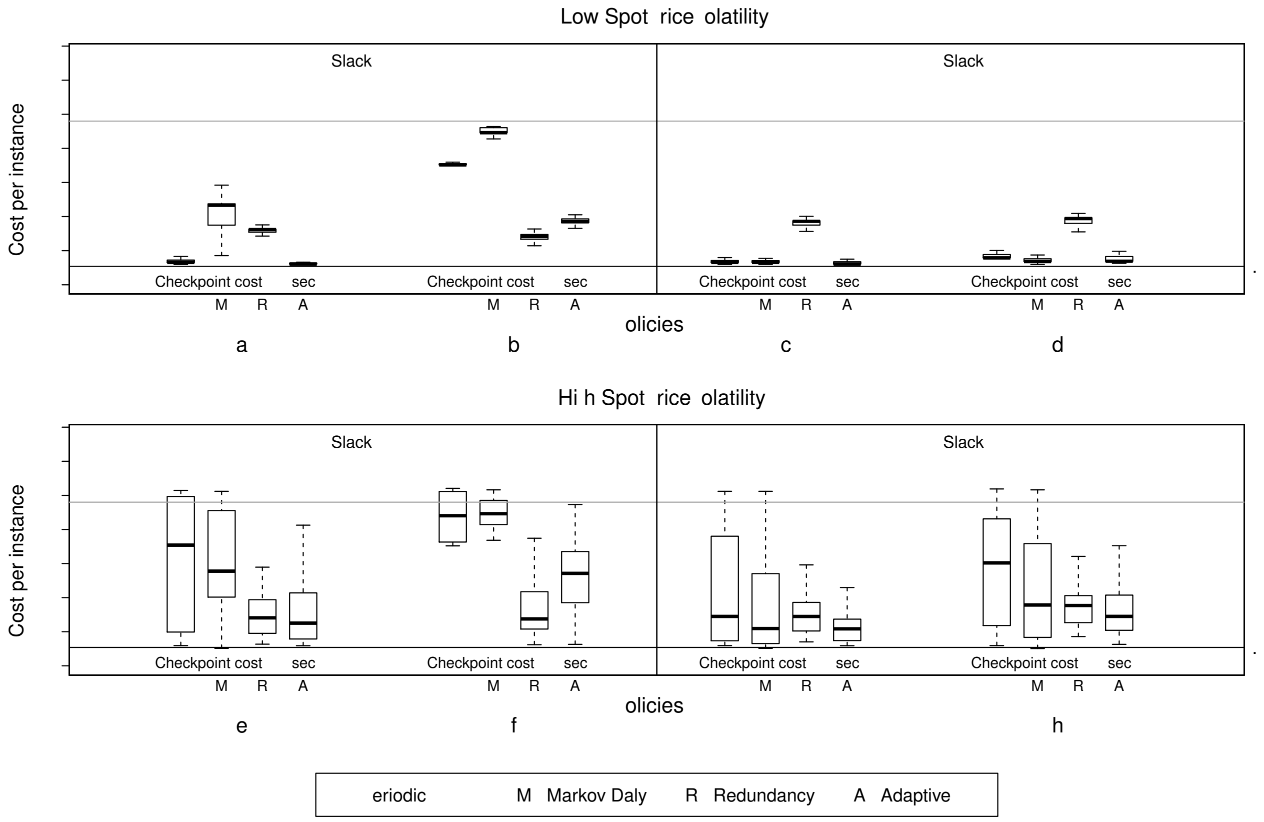

Figure 5: Comparison of <u>A</u>daptive policy with other policies: <u>P</u>eriodic, Single-zone <u>M</u>arkov-Daly and <u>R</u>edundancy-based (best-case). Results for different spot price volatility windows are shown (low at the top and high at the bottom). A horizontal grey line at \$48.00 shows the cost at on-demand rate (\$2.40/hr) and a horizontal black line at \$5.40 shows reference cost at lowest spot price (\$0.27/hr).

policy. For higher T_l values (plots (c) and (d)), the median costs of *Adaptive* are comparable to single-zone policies, but again the range of the second and third quartile costs is smaller.

With high volatility (plots (e), (f), (g) and (h)), the overall costs for *Adaptive* are strongly influenced by the amount of slack T_l. For low T_l (plots (e) and (f)), the median costs for *Adaptive* are magnified by checkpoint/restart costs. Specifically, choosing the policy that has high availability at low amortized checkpoint/restart overhead results in lower costs. Plot (e) shows that median costs for *Adaptive* are as good as the median costs for best-case redundancy-based policies. Higher checkpoint cost (t_c = 900 seconds) (plot (f)) magnifies errors, if they occur, when an incorrect policy is chosen near execution start (before there is enough information for *Adaptive* to make a good choice). This error results in higher checkpoint/restart overhead (in terms of both checkpoint frequency and cost), which inversely affects the amount of slack available on the spot market. Thus, for high checkpoint costs (t_c = 900 seconds) and low slack (T_l = 15%) for both low and high volatility windows, *Adaptive* shows higher median costs compared to best-case costs for redundancy-based policies.

For high T_l (plots (g) and (h)), *Adaptive* yields better costs, because a switch to on-demand to compensate for errors near the start of execution is not necessary (due to larger slack). Although *Adaptive* does not guarantee a total cost of less than the cost via on-demand, the upper bound on the cost is a function of the slack and (user-configurable) maximum bid price, which also applies to individual policies except *Periodic*. However, due to the way the algorithms select the policy with least predicted cost, total cost never exceeds 20% above the on-demand cost for our experiments involving 12-month data. This cost is much less than the other policies.

7.2.2 Comparison to Large-Bid

Our final comparison is *Adaptive* to *Large-bid* [8]. In *Large-bid*, the user submits a large B but maintains a second, smaller value, L, for cost control. Variable B is chosen such that it is extremely unlikely that it is ever smaller than S (e.g., B = \$100; the largest S we have observed in the 12-months data is \$20.02). With large bid, the user has no control over the cost of execution on the spot market; a price spike can result in a large expense. The motivation behind *Large-bid* is that, in general, the spot price remains significantly lower than the on-demand price. Variable L provides limited control of the cost of execution with *Large-bid*. If S moves above L, the spot instance is allowed to run for the on-going hour. If S remains larger than L near the end of the hour, a checkpoint is taken followed by a manual termination of the instance. The instance is restarted as soon as S moves below L. *Large-bid* is a strictly single zone policy. Since *Large-bid* does not provide an upper bound on

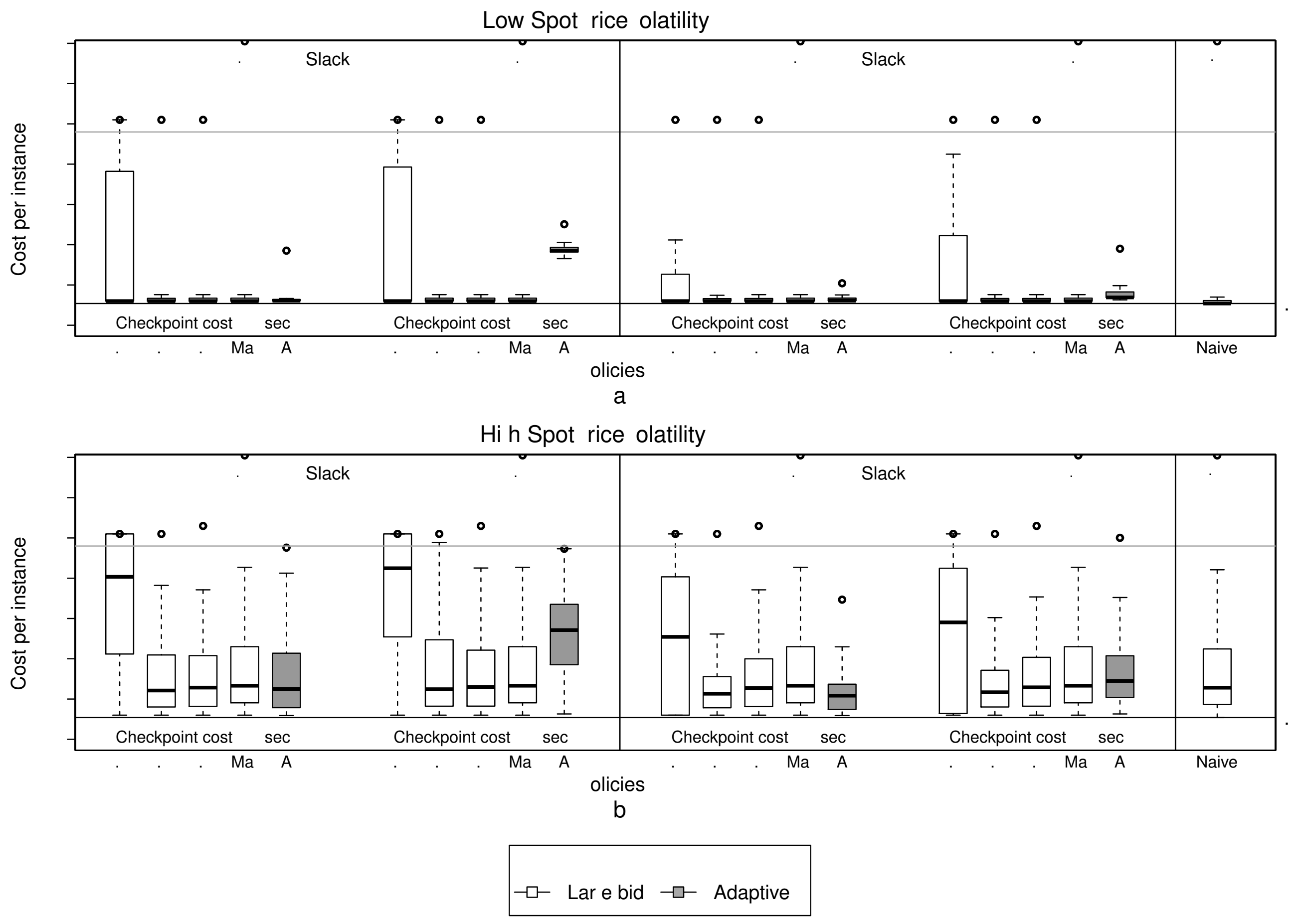

Figure 6: Comparison of *Large-bid* with *Adaptive*. Spot price volatility is low (top) and high (bottom). The user-defined threshold L is on the x-axis and the y-axis shows *Cost per Instance* in \$. The horizontal grey line at \$48.00 shows the cost at on-demand rate (\$2.40/hr) and a horizontal black line at \$5.40 shows reference cost at lowest spot price (\$0.27/hr). Circles denote maximum cost incurred.

the cost, we do not consider *Large-bid* as a candidate policy for *Adaptive*.

Figure 6 compares *Large-bid* with *Adaptive*. The user threshold is set from a low price of \$0.27 to a high price of \$20.02 (denoted as *Max* in the figure). As can be seen, in most cases *Adaptive* results in better worst-case costs than *Large-bid*. On the other hand, *Large-bid* sometimes results in lower median costs than *Adaptive* at different user thresholds for different spot price behavior and experiment configurations.

There are two key advantages of *Adaptive* over *Large-bid*. First, *Adaptive* does not incur large costs— typically well below on-demand. On the other hand, *Large-bid* offers no control over the periodic cost on the spot market until *after* the user pays a high periodic cost. For low volatility window (plot (a)), *Adaptive* results in similar median costs, but better worst-case costs than *Large-bid*. The worst-case costs for *Large-bid* are as high as *3.8x* the on-demand costs (the worst-case cost of \$183.75 results due to a spike in the spot price of \$20.02 between March 13th to March 14th, 2013). For high volatility window (plot (b)), *Adaptive* typically results in better median and worst-case costs than several *Large-bid* threshold values (except for the case where T_l = 15% and t_c = 900 seconds). The worst-case costs are as high as *2.0x* the on-demand costs (for *Max* threshold). We observe that for moderately volatile prices in a zone, *Large-bid* switches to on-demand to meet the time-bound *after* paying a high cost on the spot market. *Adaptive* predicts the optimal bid price and guarantees a *bounded* cost while completing within the time bound. Also, regardless of slack and checkpoint costs, *Adaptive* results in comparable median cost and lower worst-case cost.

Second, *Adaptive* does not have any user-chosen thresholds. In *Large-bid*, a low threshold (L = \$0.27) results in lower worst-case cost, but higher median costs. Using higher threshold values for *Large-bid* allows more resistance to checkpoint cost and slack, but increases worst-case cost. Thus, the “sweet-spot” value of the threshold depends on future spot prices that are unknown to the user. In contrast, *Adaptive* handles this implicitly, with no input from the user. Importantly, when no threshold is used (labeled *Naive* in the figure), the worst case *Large-bid* cost is larger than the worst-case cost for *Adaptive*. Moreover, without a threshold, reaching this worst-case cost in a given experiment is more likely, even for a situation with moderate spot price volatility.

7.3 Summary

Adaptive shows median costs competitive to best-case median costs for existing single-zone policies. Choosing the policy with least predicted cost for high spot price volatility is non-trivial at low slack (T_l). An error in making this choice is magnified by higher checkpoint/restart costs (t_c). Even for high spot price volatility, *Adaptive* results in median costs better than existing single-zone policies and competitive to best-case costs for redundancy-based policies for a configuration with low T_l and high t_c. For other configurations, *Adaptive* results in median costs similar to the best-case median costs of the other three policies. Unlike *Large-bid*, *Adaptive* chooses the user bid and the policy with the least predicted cost resulting in a *bounded* cost, even with high spot price volatility.

8. RELATED WORK

The problem of optimizing cost of running HPC applications on the cloud has been an active area of research. Previous work focuses on predicting spot price movements for selecting the optimal bid price and fault-tolerance technique on the EC2 spot market. Machine learning approaches to predict future spot prices apply well-known statistical models to study spot price distribution [6, 1, 10]. Chohan et. al employ a Markov model to predict instance up time [2] for MapReduce-type applications.

There exists a large body of work on exploring bid price prediction and fault-tolerance strategies. Yi et. al present cost-performance trade-offs of different checkpoint scheduling policies on the spot market [18]. The study shows two things: that the frequency of checkpointing directly affects total execution time of the application, and that the frequency of checkpointing affects the time to recover from failure. Therefore, the choice of frequency also affects total cost of execution, as higher overhead or higher recovery time both contribute to monetary cost. Yi et. al and Voorsluys et. al extend the preceding work to explore cost-effectiveness of different cost-aware checkpointing schemes coupled with task migration and duplication on to different resource types [17, 16]. Since we address tightly coupled HPC applications running on HPC-grade CC2 instances, addressing their work is beyond the scope this paper. Jung et. al present an improved *Edge* algorithm to efficiently schedule checkpoints [7]. Another scheme presented by Khatua et. al presents the large-bid approach for cost-effective runs on the spot market [8]. We address both schemes in our evaluation. Our work differs from previous work in two ways: (1) we evaluate redundancy as the first-class fault-tolerance mechanism at different bid prices, and (2) our work provides guarantee on job completion times.

Previous work on optimizing cost of time-constrained execution on the spot market predicts optimal bid prices for each hourly billing cycle [12, 19] with a fixed, hourly checkpoint frequency. Work by Tang et. al [12] focuses on optimizing cost or performance with time or cost constraints, respectively. Their work does not guarantee a strict deadline and cannot be directly applied to tightly coupled HPC applications. Work by Zafer et. al [19] addresses the problem of running loosely coupled applications and does not predict optimal checkpoint frequency. Our *Adaptive* checkpoint scheduling scheme solves the problem of running tightly coupled HPC applications with a guaranteed completion time and predicts the optimal bid price, number of zones and optimal checkpoint frequency for each billing cycle.

9. CONCLUSION

This paper focused on exploiting redundancy for cost-effective execution on the spot market. The user provides an execution time bound, and our *Adaptive* algorithm chooses a bid price and a checkpoint-insertion algorithm that results in meeting the bound at low cost. We found that in comparison to *Large-bid*, *Adaptive* has bounded costs and avoids worst cases in which the user is charged an exceedingly large amount (e.g., more than using the on-demand market). Overall, we believe that our *Adaptive* scheme is a step towards more practical, cost-effective use of the spot market.

APPENDIX

A. SPOT MARKET MODIFICATIONS

In this appendix we explain in detail the ramifications of modifications to the spot market mechanism. First, instead of abrupt termination, the user could be provided with a notification that the spot price is about to change and the instance may be terminated. In this case, the user could take a checkpoint and save application state just before termination. However, we argue that such a window would not always be sufficiently large to save the state of an HPC application with a large working set over a large number of tasks (the checkpoint costs also involve bottleneck at the I/O server–*ten* minutes). On the other hand, a window of notification sufficiently large to save a checkpoint would work against the principle of the spot market in which prices change within ten minutes.

Second, the user could be allowed to adjust the user bid at run time to retain spot instances at high spot prices. However, there are several problems with such a provision. First, spot prices primarily consist of spikes that are difficult to predict (from our study), thus necessitating the use of a fault-tolerance mechanism. Second, a scheme in which the user follows the spot price essentially becomes a "large-bid" policy described in Section 7, which is inferior in worst-case costs. Third, for long periods of time where spot prices are predictable, the provision may enable the user to utilize the spot market at a much cheaper cost, disrupting the balance between the spot and on-demand markets. This would lower Amazon's overall profit.

Third, when the spot price is about to rise over the user bid price, Amazon could offer the user a significantly higher rate for shorter billing cycles (e.g. 5-minute cycle), during which the application state could be saved. Such an enhancement seems better for the user since the user could pay more to avoid the need to employ fault tolerance. It also seems better for Amazon, who could attract more users who would like to avoid the need for fault tolerance. However, we find the following issues with the provision. First, it would be difficult to apply pricing at such a short billing granularity and hard to predict (for both the user and Amazon) how long it will take to save the application state. Second, the price for the short billing cycle would likely be variable, since longer uninterrupted runs for the user could mean more "value" associated with the checkpoint. Third, such a scheme would work against the second objective of the spot market–of keeping the users from hogging low-cost resources for a long time. Fourth, it would also work against free partial-hour usage, which is attractive for users with short-lived burst requests. We argue that the above reasons would prevent Amazon from making such provisions in the spot market mechanism.

B. MARKOV MODEL DETAILS

For completeness, we provide the basic idea behind the Markov model. We calculate the expected up time of the instance at a bid price B as follows: $PROB$ denotes a $1 \times N$ probability matrix for N different spot prices in the price history. $TRANS$ is an $N \times N$ transition matrix in which the element at (n, m) represents the probability of spot price moving from n to m, $\forall\ n, m \in N$ (i.e. n^{th} row to m^{th} column). For a step size of 5 minutes, we predict the state of $PROB$ for the next step as shown in Equation 2. We choose step size of 5 minutes because spot price movements do not occur in such duration in most cases (from our price history of 12 months).

$$\forall j \in N, PROB_j^{k+1} = \sum_{i=1}^{N} [PROB_j^k \times TRANS_{i,j} \times I(i)] \quad (2)$$

$$I(i) = \begin{cases} 1, & \text{if } P_i \leqslant B \\ 0, & \text{otherwise} \end{cases}$$

Expected up time $E[T_u]$ is calculated as a weighted average over k=1 to Th steps as shown in Equation 3. Intuitively, the equation calculates a weighted average of zone up-time over each step with conditional probability of zone being terminated at the end of each step (note that the condition is reversed for $I(i)$ [not shown]). Th is the minimum value at which the expected up time does not change at seconds granularity over multiple iterations.

$$E[T_u] = \sum_{k=1}^{Th} k \times \left[\sum_{i=1}^{N} PROB_i^k \times I(i) \right] \quad (3)$$

Acknowledgments

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-652718). We also thank Amazon for a grant for time on EC2. Finally, we thank the anonymous reviewers for comments that improved the quality of this paper.

C. REFERENCES

[1] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 304–311, 2011.

[2] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See spot run: using spot instances for mapreduce workflows. In *USENIX Hot Topics in Cloud Computing*, pages 7–7, 2010.

[3] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, pages 303–312, Feb. 2006.

[4] Gartner Inc. Toolkit: Comparison Matrix for Cloud Infrastructure as a Service Providers, 2013. `https://www.gartner.com/doc/2575815`, 2013.

[5] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[6] B. Javadi, R. K. Thulasiramy, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 219–228, 2011.

[7] D. Jung, S. Chin, K. Chung, H. Yu, and J. Gil. An efficient checkpointing scheme using price history of spot instances in cloud computing environment. In *Proceedings of the 8th IFIP International Conference on Network and Parallel Computing*, 2011.

[8] S. Khatua and N. Mukherjee. Application-centric resource provisioning for Amazon EC2 spot instances. In *International Conference on Parallel Processing*, pages 267–278, 2013.

[9] M. Liu, Y. Jin, J. Zhai, Y. Zhai, Q. Shi, X. Ma, and W. Chen. ACIC: Automatic Cloud I/O Configurator for HPC applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO*, page 38, 2013.

[10] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *IEEE International Conference on High Performance Computing and Communications*, pages 296–303, 2011.

[11] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen. Cost-effective cloud HPC resource provisioning by building semi-elastic virtual clusters. In *ACM/IEEE Supercomputing*, page 12, 2013.

[12] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *IEEE International Conference on Cloud Computing*, pages 91–98, 2012.

[13] A. Technica. $4,829-per-hour supercomputer built on Amazon cloud to fuel cancer research. `http://arstechnica.com/business/news/2012/04/4829-per-hour-supercomputer-built-on-amazon-cloud-to-fuel-cancer-research.ars`, 2012.

[14] A. Technica. 18 hours, $33K, and 156,314 cores: Amazon cloud HPC hits a "petaflop". `http://arstechnica.com/information-technology/2013/11/18-hours-33k-and-156314-cores-amazon-cloud-hpc-hits-a-petaflop/`, 2013.

[15] TOP500. Top500 List - November 2013. `http://top500.org/list/2013/11/`, 2013.

[16] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *IEEE Int'l Conference on Advanced Information Networking and Applications*, pages 542–549, 2012.

[17] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 2011.

[18] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In *IEEE International Conference on Cloud Computing*, pages 236–243, 2010.

[19] M. Zafer, Y. Song, and K.-W. Lee. Optimal bids for spot VMs in a cloud for deadline constrained jobs. In *IEEE International Conference on Cloud Computing*, pages 75–82, 2012.