Online Risk Analytics on the Cloud

Hyunjoo Kim, Shivangi Chaudhari, Manish Parashar*, and Christopher Marty†
*NSF Center for Autonomic Computing
Department of Electrical & Computer Engineering
Rutgers, The State University of New Jersey
94 Brett Road, Piscataway, NJ, USA
Email: {hyunjoo, shivangc, parashar}@rutgers.edu
†Bloomberg LP, New York, NY, USA
Email: cmarty@bloomberg.net

***Abstract*—In todays turbulent market conditions, the ability to generate accurate and timely risk measures has become critical to operating successfully, and necessary for survival. Value-at-Risk (VaR) is a market standard risk measure used by senior management and regulators to quantify the risk level of a firm's holdings. However, the time-critical nature and dynamic computational workloads of VaR applications, make it essential for computing infrastructures to handle bursts in computing and storage resources needs. This requires on-demand scalability, dynamic provisioning, and the integration of distributed resources. While emerging utility computing services and clouds have the potential for cost-effectively supporting such spikes in resource requirements, integrating clouds with computing platforms and data centers, as well as developing and managing applications to utilize the platform remains a challenge. In this paper, we focus on the dynamic resource requirements of online risk analytics applications and how they can be addressed by cloud environments. Specifically, we demonstrate how the CometCloud autonomic computing engine can support online multi-resolution VaR analytics using and integration of private and Internet cloud resources.**

I. Introduction

In todays turbulent market conditions, financial firms must carefully monitor and manage risk or face severe consequences up to and including bankruptcy. Accordingly, the ability to generate accurate and timely risk measures has become critical to operating successfully, and necessary for survival. Value-at-Risk (VaR) is a market standard risk measure used by senior management and regulators to quantify the risk level of a firm's holdings. The VaR measure looks at the entirety of the firm's holdings at a confidence interval and time horizon, and reports an expected loss number. For example, a 1-Day 99% VaR number of $1 Million means that with 99% confidence, the firm holdings won't decrease in value by more than $1 Million over the next day. The non-linear nature of instrument pricing models and the requirement to preserve correlations of price movements make developing a closed form solution to this problem virtually impossible for all but the most trivial portfolios. Large complex VaR calculations are typically done using computationally intensive Monte-Carlo simulations.

Consider a medium size firm holding positions in 20,000 different financial instruments. Running a 100,000 simulation Monte-Carlo VaR calculation requires generating 2 Billion simulated instrument prices. With a conservative estimate of 10 ms. per pricing, this calculation requires more than 5,500 hours of processor time or roughly 700 processors working concurrently over an 8-hour window. As a result, the capital cost of hardware plus the operational cost for data center space, power, cooling, and maintenance make this cost prohibitive to all but the largest firms. The tradeoff between increased cost and complexity versus careful and accurate risk measures have driven financial firms to look for innovative ways to decrease computing costs and, at the same time, increase quality of risk measurements.

There are a number of properties of the VaR calculation that make it a compelling candidate for a cloud computing architecture. A VaR calculation will typically start after the end of the trading day, when market data and final positions have been verified. It must complete, and updated risk numbers must be available, before the start of the next trading day. As the number and complexity of positions change, the computational requirements for the calculation can change significantly, however the completion deadline of the beginning of the next trading day remains fixed. Furthermore, as market conditions change, a firm may want to vary the number of Monte Carlo scenarios run (and thus the resolution of the calculation), which will add additional variability to the computation time. The requirement for additional computation happens irregularly. Besides, considering the significant computational resources necessary for VaR, statically over provisioning computing resources to account for worst-case requirements does not make economic sense. As a result, the elastic nature of cloud computing resources makes it a very attractive solution.

Additionally, the IT staff of a typical financial institution lacks the specialized development and operational resources necessary to build and operate a large scale distributed processing environment. The ability to outsource this complexity, as well as to take advantage of favorable pricing due to the off hours nature of the VaR calculation is very compelling. How-

The research presented in this paper is supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826, and CNS 0723594, and by Department of Energy via the grant number DE-FG02-06ER54857, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University.

The research is also supported by Amazon Elastic Compute Cloud.

ever, integrating clouds with in-house computing platforms and data centers to support dynamic on-demand provisioning and scale-out, as well as developing and managing applications to utilize the platform remains a challenge.

The goal of this paper is twofold: (1) to investigate the feasibility of using cloud computing services to support the dynamic requirements of online risk analytics, as well as (2) to demonstrate the ability of the CometCloud autonomic computing engine to provide programming and runtime infrastructure support to enable these applications to seamlessly and safely scale-out (and scale-in) from in-house private datacenters to Internet clouds such as the Amazon EC2, based on the dynamic computational load. Specifically, in this paper we demonstrate how the CometCloud autonomic computing engine can support on-demand multi-resolution VaR analytics using cloud resources. A prototype implementation as well as an initial evaluation using Amazon EC2 as well as an internal Rutgers cloud platform are presented.

The rest of this paper is organized as follows. Section II presents an overview of VaR and Section III describes the CometCloud framework for autonomic cloud bursts using Comet. Section IV presents the design and implementation of the VaR using CometCloud. An evaluation of the CometCloud is presented in Section V and a conclusion is in Section VI.

II. Overview of Value-at-Risk Analytics

Monte-Carlo VaR is a very powerful measure used to judge the risk of portfolios of financial instruments. The complexity of the VaR calculation stems from simulating portfolio returns. To accomplish this, Monte-Carlo methods are used to "guess" what the future state of the world may look like. Guessing a large number of times allows the technique to encompass the complex distributions and the correlations of different factors that drive portfolio returns into a discreet set of *scenarios*. Each of these Monte-Carlo scenarios contains a state of the world comprehensive enough to value all instruments in the portfolio, and thus allows us to calculate a return for the portfolio under that scenario.

The process of generating Monte-Carlo scenarios begins by selecting primitive instruments or *invariants*. To simplify simulation modeling, invariants are chosen such that they exhibit returns that can be modeled using a stationary normal probability distribution [1]. In practice these invariants are returns on stock prices, interest rates, foreign exchange rates, etc. The universe of invariants must be selected such that portfolio returns are driven only by changes to the invariants.

The goal of generating Monte-Carlo scenarios is to project forward many possible realizations of invariant values at the current time horizon. To do this, the dispersion of individual invariant returns and the correlation between risk factor returns must be examined. The time series of invariant returns is used to create a *covariance* matrix. A covariance matrix (typically represented by the symbol Σ) is a square matrix of size N, where N represents the number of invariants. Terms on the diagonal represent the variance of invariants, and terms off the diagonal represent the co-variance of pairs of invariants.

In the trivial case where volatility of the portfolio is a linear function of the volatility of the invariants, techniques can be used to scale Σ by the linear weighting of the invariants to calculate a portfolio variance. For example, if the constituents of a portfolio are the two stocks IBM and DELL, we can select the returns of these two stocks as invariants and, based on their historical returns, create a 2×2 covariance matrix Σ. Based in the weights of IBM and DELL in the portfolio, we can create a vector of weights w, and the resulting variance of the portfolio is $\sigma^2_{portfolio} = w'\Sigma w$. By extending the assumption of normal returns and using the inverse normal at 0.99, we can calculate the 99% 1-Day VaR as $2.33 \times \sigma^2_{portfolio}$. This parametric approach [2] to calculating VaR is useful if the constituents of a portfolio exhibit linear returns to risk factor changes.

Stock options are a type of instrument where volatility is a non-linear function of the invariants. Simple stock options can be classified into *puts* and *calls*. A call option is the right, but not the obligation to purchase a stock at a given price K, while a put option is the right, but not the obligation, to sell a stock at a given price K (strike price). If we take S to be the current price of a stock, the value of a call option when exercised is $\max\{0, K - S\}$, while the value of a put option when exercised is $\max\{0, S - K\}$. With an European style option, your option to buy the stock at price K can only be exercised at the expiration (or end) of the contract. An American style option can be exercised at any point between the purchase and expiration of the option contract.

The value of the option at a point in time is a function of the strike price K, the current stock price S, the volatility of the stock price, interest rates, and the expiration date of the contract. This value is a non-linear function of the inputs. European style options are priced using the Black-Scholes formula. Though imperfect in practice, the Black-Scholes formula is a staple of option pricing and is well documented in other sources [3]. There is currently no closed form solution for pricing American style options. Options of this class are priced in a number of ways including using risk-neutral expectations on a binary tree. This method of option valuation is well known and documented in [4].

To properly capture the nonlinear pricing of portfolios containing options, we use Monte-Carlo techniques to simulate many realizations of the invariants. Each realization is referred to as a scenario. Under each of these scenarios, each option is priced using the invariants and the portfolio is valued. As outlined above, the portfolio returns for scenarios are ordered from worst loss to best gain, and a VaR number is calculated.

III. CometCloud and Autonomic Cloud Bursts

A. CometCloud architecture

We have developed the CometCloud autonomic computing engine, based on Comet [5], to support autonomic cloud bursts and the motivating scenarios described in the following subsection. A schematic overview of the architecture is presented in Fig. 1. CometCloud, like Comet, is based on a peer-to-peer substrate and runs on enterprise datacenters, Grids and clouds.

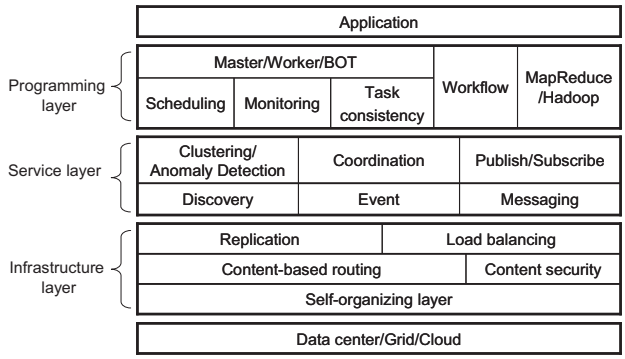


Fig. 1. The CometCloud architecture for autonomic cloud bursts

Conceptually, CometCloud is composed of a programming layer, service layer, and infrastructure layer. The infrastructure layer uses the Chord self-organizing overlay [6], and the Squid [7] information discovery and content-based routing substrate build on top of Chord. The routing engine [8] supports flexible content-based routing and complex querying using partial keywords, wildcards, or ranges. It also guarantees that all peer nodes with data elements that match a query/message will be located. Note that resources (nodes) in the overlay have different roles (and accordingly, access privileges) based on their credentials and capabilities. This layer also provides replication and load balancing services, and handles dynamic joins and leaves of nodes as well as node failures. Every node keeps the replica of its successor node's state, and reflects changes to this replica whenever its successor notifies it of changes. If a node fails, the predecessor node merges the replica into its state and then make a replica of its new successor. If a new node joins, the joining node's predecessor updates its replica to reflect the joining node's state, and the successor gives its state information to the joining node. Note that to maintain load balancing, load should be redistributed among the nodes whenever a node joins and leaves. Further details about the Comet substrate can be found in [5].

The service layer provides a range of services to supports autonomics at the programming and application level. This layer supports the Linda-like [9] tuple space coordination model, and provides a virtual shared-space abstraction as well as associative access primitives. Dynamically constructed transient spaces are also supported to allow the applications explicitly exploit context locality for improving system performance. Asynchronous (publish/subcribe) messaging and evening services are also provides by this layer. Finally, online clustering services support autonomic management and enable self-monitoring and control. Events describing the status or behavior of system components are clustered and the clustering is used to detect anomalous behaviors.

The programming layer provides the basic framework for application development and management. It supports a range of paradigms including the master/worker/BOT. Masters generate tasks and workers consume them. Masters and workers can communicate via virtual shared space or using a direct connection. Scheduling and monitoring of tasks are supported by the application framework. The task consistency service handles lost tasks. Even though replication is provided by the infrastructure layer, a task may be lost due to network congestion. In this case, since there is no failure, infrastructure level replication may not be able to handle it. This can be handled by the master, for example, by waiting for the result of each task for a pre-defined time interval, and if it does not receive the result back, regenerating the lost task. If the master receives duplicate results for a task, it selects one (the first) and ignores other (subsequent) results. Other supported paradigms include workflow-based applications as well as Mapreduce [10] (and Hadoop [11]).

B. Autonomic Cloud Bursts

The goal of autonomic cloud bursts is to seamlessly (and securely) integrate private enterprise clouds and datacenters with public utility clouds on-demand, to provide and abstraction of resizable computing capacity. It enables the dynamic deployment of application components, which typically runs on internal organizational compute resources, onto a public cloud to address dynamic workloads, spikes in demands, and other extreme requirements. Furthermore, given the increasing application and infrastructure scales, as well as their cooling, operation and management costs, typical over-provisioning strategies are no longer feasible. Autonomic cloud bursts can leverage utility clouds to provide on-demand scale-out and scale-in capabilities based on a range of metrics. Key motivations for autonomic cloud bursts in the context of risk analytics applications include:

- *Load dynamics:* VaR application workloads can vary significantly. This includes the number of application tasks as well the computational requirements of a task. The computational environment must dynamically grow (or shrink) in response to these dynamics while still maintaining strict deadlines.
- *Accuracy of the analytics:* The required accuracy of the risk analytics depends on a number of highly dynamic market parameters, and has a direct impact on the computational demand, e.g., the number of scenarios in the Monte-Carlo VaR formulation. The computational environment must be able to dynamically adapt to satisfy the accuracy requirements while still maintaining strict deadlines.
- *Economics:* Application tasks can have very heterogeneous and dynamic priorities, and must be assigned resources and scheduled accordingly. Budgets and economic models can be used to dynamically provision computational resources based on the priority and criticality of the application task. For example, application tasks can be assigned budgets and can be assigned resources based on this budget. The computational environment must be able to handle heterogenous and dynamic provisioning and scheduling requirements.

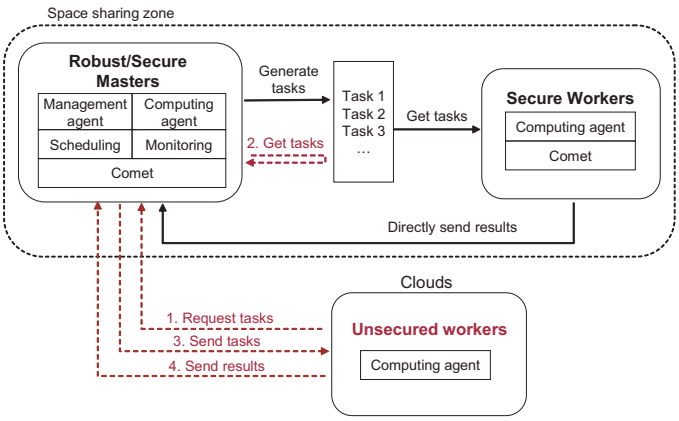


Fig. 2. Support for autonomic cloud bursts in CometCloud.

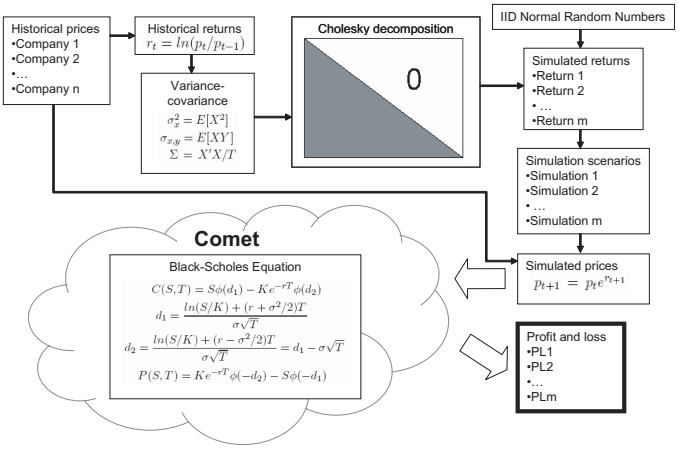


Fig. 3. Workflow for the distributed VaR computation using CometCloud

- *Failures:* Due to the strict deadlines involved, failures can be disastrous. The computation must be able to manage failures without impacting application quality of service, including deadlines and accuracies.

The overall approach for support autonomic cloud bursts in CometCloud is presented in Fig. 2. CometCloud considers three types of clouds based on perceived security/trust and assigns capabilities accordingly. The first is a highly trusted, robust and secure cloud, usually composed of trusted/secure nodes within an enterprise, which is typically used to host masters and other key (management, scheduling, monitoring) roles. These nodes are also used to store state. In most applications, the privacy and integrity of critical data must be maintained, and as a result, task involving critical data should be limited to cloud nodes that have required credentials. The second type of cloud is one composed of nodes with such credentials, i.e., the cloud of secure workers. A privileged Comet space may span these two clouds and may contain critical data, tasks and other aspects of the application-logic/workflow. The final type of a cloud consists of casual workers. These workers are not part of the space but can access the space through the master to obtain (possibly encrypted) work units as long as they present required credentials. Note that while nodes can be added or deleted from any of these clouds, autonomic cloud bursts primarily target workers nodes, and specifically worker nodes that do not host the Comet space as they are less expensive to add and delete.

IV. Value-at-Risk on CometCloud

This section describes the parallel VaR implementation using CometCloud. The overall distributed VaR workflow is presented in Fig. 3. The algorithm starts by loading the historical input prices, computes the historical returns and the variance-covariance matrix, decomposes the variance matrix into the Cholesky factorization, simulates the Monte-Carlo scenarios, i.e., simulated prices, and computes the simulated returns and prices. Using the simulated prices, the master orchestrates the execution of the pricing algorithms in parallel on available workers, and computes the profits and losses based on the results produced by the workers. The workers execute the option pricing algorithm, i.e., solve the Black-Scholes equations for the European pricing model on independent subsets of the simulated prices. The algorithm for the worker transfers a subset of simulated prices for one Monte-Carlo scenario, runs the pricing algorithm on this subset, and sends the results back to the master.

The flowchart in Fig. 4 presents the overall flow of VaR application using the CometCloud infrastructure. When CometCloud is initiated, a bootstrap node starts first and the Chord overlay is formed as nodes join. The CometCloud shared space is built integrating local spaces at nodes with appropriate credentials. The master and worker modules are then configured. Once the computation starts, the master creates tasks and inserts them into the space using the *"out"* primitive and workers get tasks from the space using the *"in"* primitive and consume them. There are two sets of tasks generated in the application implemented. The first set is to get simulated returns used to compute simulated prices. The second set is for the P&L and VaR calculations. A typical *out* task is described by an XML tag as shown below.

```
<VarAppTask>
   <TaskId>taskid</TaskId>
   <DataBlock>data_blocks</DataBlock>
   <MasterId>masterId</MasterId>
   <MasterNetName>masterNetName</MasterNetName>
</VarAppTask>
```

The task is then converted into the tuple object and the data is stored in the data attribute of the tuple in the form of serialized bytes. The worker uses the *in* function to query for the task in the space, reads the tuple as well as data to do the required calculations. It then sends back the results directly to the master after calculations.

V. Evaluation

In this section, we present an initial experimental evaluation of CometCloud. This includes an evaluation of the CometCloud operations, an evaluation of the CometCloud-

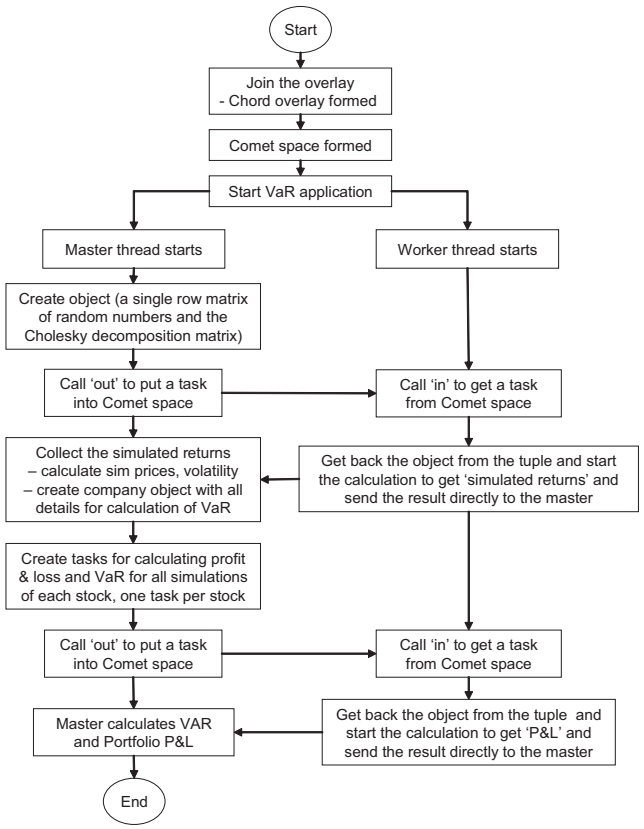
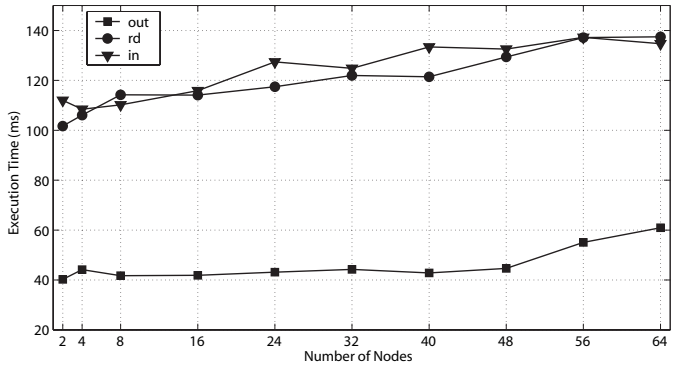
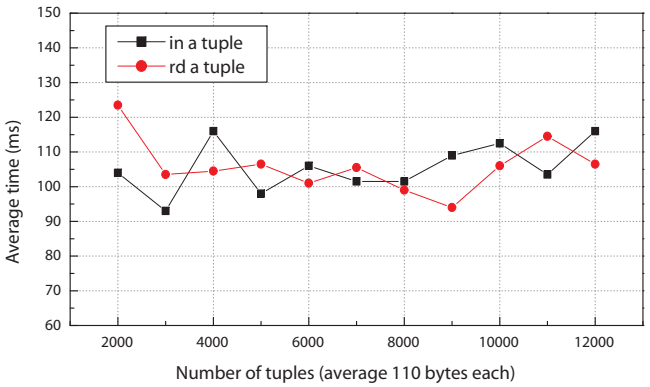


Fig. 4. The overall flow of the VaR application using CometCloud



(a) Average time for *out*, *in*, and *rd* operators for increasing system sizes



(b) Average time for *in* and *rd* operations with increasing number of tuples. System size fixed at 4 nodes.

Fig. 5. Evaluation of CometCloud primitives on the Rutgers cloud.

based VaR application on Amazon EC2, and an evaluation of policy driven autonomic cloud bursts in CometCloud.

Evaluation of basic CometCloud operations: In this experiment we evaluated the costs of basic tuple insertion and exact retrieval operations on the Rutgers cloud. Each machine was a peer node in the CometCloud overlay and the machines formed a single CometCloud peergroup. The size of the tuples in the experiment were fixed at 200 bytes. A ping-pong like process was used in the experiment, in which an application process inserted a tuple into the space using the *out* operator, read the same tuple using the *rd* operator, and deleted it using the *in* operator. In the experiment, the *out* and exact matching *in/rd* operators used a 3-dimension information space. For an *out* operation, the measured time corresponded to the time interval between when the tuple was posted into the space and when the response from the destination was received. For an *in* or *rd* operation, the measured time was the time interval between when the template was posted into the space and when the matching tuple was returned to the application, assuming that a matching tuple existed in the space. This time included the time for routing the template, matching tuples in the repository, and returning the matching tuple. The average performances were measured for different system sizes.

Fig. 5 (a) plots the average measured performance and shows that the system scales well with increasing number of peer nodes. When the number of peer nodes increases 32 times, i.e., from 2 to 64, the average round trip time increases only about 1.5 times, due to the logarithmic complexity of the routing algorithm of the Chord overlay. *rd* and *in* operations exhibit similar performance, as shown in the figure. To further study the *in/rd* operator, the average time for *in/rd* was measured using increasing number of tuples. Fig. 5 (b) shows that the performance of *in/rd* is largely independent of the number of tuples in the system – the average time is approximately 105 ms as the number of tuples is increased from 2,000 to 12,000.

Evaluation of the CometCloud-based VaR application on Amazon EC2: Fig. 6 presents the total application runtime of a CometCloud-based VaR application on Amazon EC2 for different number of VaR simulations. In this experiment, we ran a master on the Rutgers cloud and up to five workers on EC2 instances. Each worker ran a different instance. In the experiment, we assumed that all workers were secure and were part of the CometCloud space. As shown in the figure, and as expected, the application runtime decreases as the number of EC2 workers increase and the scalability improves as the number of simulations (or workload) increases.

Evaluation of autonomic cloud bursts in CometCloud: In this experiment, autonomic cloud bursts is represented by the number of changing workers. When the application workload increases (or decreases), a pre-defined number of workers

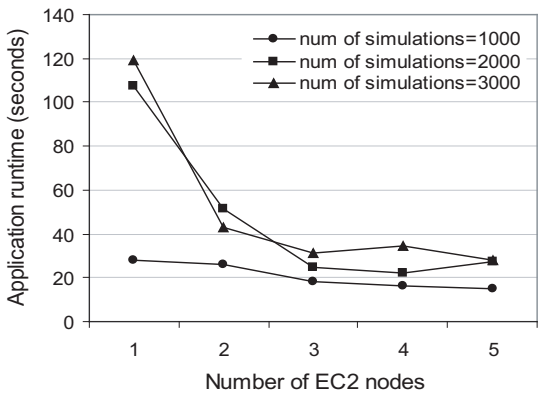


Fig. 6. Evaluation of the CometCloud-based VaR application on Amazon EC2.

are added (or released), based on the application workload. The autonomic resource provisioning is based on predefined policies. We define four types of policies.

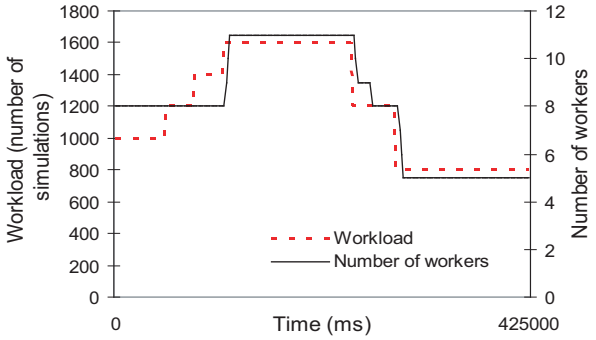
- *Time-based:* When an application needs to be completed as soon as possible, assuming an adequate budget, the maximum required workers are allocated for the job.
- *Budget-based:* When a budget is enforced on the application, the number of workers allocated must ensure that the budget is not violated.
- *Workload-specific:* In this policy, when the application workload increases or decreases, the number of allocated or released is explicitly defined by the appplication.
- *Bounded-workload:* In this policy, whenever the application the workload increase by more than a specified threshold a predefined number of workers is added. Similarly, if the workload decreases by more that the specified threshold the predefined number of workers a released.

Fig. 7 demonstrate autonomic cloud-bursts in CometCloud based on two of the above polices, i.e., workload-specific and bounded-workload. The figures plots the changes in the number of worker as the workload changes. For the workload-specific policy, the initial workload is set to 1000 simulations and the initial number of workers is set to 8. The workload is then increased or decreased workload by 200 simulations at a time and the number of worked added or released set to 3. For bounded-workload policy, the number of workers is initially 8 and the workload is 1000 simulations. In this experiment, the workload is increased by 200 and decreased by 400 simulations, and 3 workers are added or released at a time. The plots in Fig. 7 clear demonstrate the cloud burst behavior. Note that the policy used as well as the thresholds can be changed on-the-fly.

VI. Conclusion and Future Work

In this paper, we presented a cloud-based implementation of online advanced risk-analytics, and specifically, the VaR application, using the CometCloud autonomic computing engine. The goal was to study the feasibility and applicability of autonomic cloud bursts to cost-effectively support the dynamic requirements of risk analysis applications in computational finance. We deployed the VaR application on the Rutgers Cloud and Amazon EC2 and presented an initial performance evaluation as well as demonstrated policy-based autonomic cloud brusts.

(a) Workload-specific policy



(b) Bounded-workload policy

Fig. 7. Policy-based Autonomic cloud burst using CometCloud.

References

[1] A. Meucci, *Risk and Asset Allocation*. New York, NY: Springer-Verlag, 2005.
[2] J. Mina and J. Xiao, "Return to riskmetrics: The evolution of a standard," RiskMetrics Group, Tech. Rep., 2001.
[3] M. Baxter and A. Rennie, *Financial Calculus*. Cambridge, U.K.: Cambridge University Press, 1996.
[4] S. E. Shreve, *Stocastic Calculus for Finance 1: Binomial Asset Pricing Model*. New York, N.Y.: Springer-Verlag, 2004.
[5] Z. Li and M. Parashar, "A computational infrastructure for grid-based asynchronous parallel applications," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2007, pp. 229–230.
[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," 2001, pp. 149–160.
[7] C. Schmidt and M. Parashar, "Squid: Enabling search in dht-based systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 7, pp. 962–975, 2008.
[8] ——, "Enabling flexible queries with guarantees in p2p systems," *IEEE Internet Computing*, vol. 8, no. 3, pp. 19–26, 2004.
[9] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.
[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[11] Hadoop, "http://hadoop.apache.org/core/."