Model-driven Auto-scaling of Green Cloud Computing Infrastructure

Brian Dougherty[a], Jules White[b], Douglas C. Schmidt[a]

[a]*Institute for Software Integrated Systems, Vanderbilt University, Campus Box 1829 Station B, Nashville, TN 37235, Email:{briand,schmidt}@dre.vanderbilt.edu*
[b]*ECE, 302 Whitemore Hall, Virgnia Tech, Blacksburg, VA 24060, Email:julesw@vt.edu*

Abstract

Cloud computing can reduce power consumption by using virtualized computational resources to provision an application's computational resources on-demand. Auto-scaling is an important cloud computing technique that dynamically allocates computational resources to applications to match their current loads precisely, thereby removing resources that would otherwise remain idle and waste power. This paper presents a model-driven engineering approach to optimizing the configuration, energy consumption, and operating cost of cloud auto-scaling infrastructure to create greener computing enviornments that reduce emissions resulting from superfluous idle resources. The paper provides four contributions to the study of model-driven configuration of cloud auto-scaling infrastructure by (1) explaining how virtual machine configurations can be captured in feature models, (2) describing how these models can be transformed into constraint satisfaction problems (CSPs) for configuration and energy consumption optimization, (3) showing how optimal auto-scaling configurations can be derived from these CSPs with a constraint solver, and (4) presenting a case-study showing the energy consumption/cost reduction produced by this model-driven approach.

1 Introduction

Current trends and challenges. By 2011, power consumption of computing data centers is expected to exceed 100,000,000,00 kilowatt-hours(kWh) and generate over 40,568,000 tons of CO_2 emissions [1, 2, 3]. Since data centers operate at only 20-30% utilization, 70-80% of this power consumption is lossed due to over-provisioned idle resources, resulting in roughly 29,000,000 tons of unnecessary CO_2 emissions [1, 2, 3]. Applying new computing paradigms, such as cloud computing with auto-scaling, to increase server utilization and decrease idle time is therefore paramount to creating greener computing environments with reduced power consumption and emissions [4, 5, 6, 7, 8].

Cloud computing is a computing paradigm that uses virtualized server infrastructure and auto-scaling to provision virtual OS instances dynamically [9]. Rather than

Email address: `julesw@vt.edu` (Jules White) over-provisioning an application's infrastructure to meet peak load demands, an application can *auto-scale* by dynamically acquiring and releasing virtual machine (VM) instances as load fluctuates. Auto-scaling increases server utilization and decreases idle time compared with over-provisioned infrastructures, in which superfluous system resources remain idle and unneccesarily consume power and emit superfluous CO_2. Moreover, by allocating VMs to applications on demand, cloud infrastructure users can pay for servers incrementally rather than investing the large up-front costs to purchase new servers, reducing up-front operational costs.

Although cloud computing can help reduce idle resources and negative environemntal impact, running with less instantly available computing capacity can impact quality-of-service (QoS) as load fluctuates. For example, a prime-time television commercial advertising a popular new product may cause a ten-fold increase in traffic to the advertisers website for about 15 minutes. Data centers can use existing idle resources to handle this momentary increase in demand and maintain QoS. Without these additional resources, the website's QoS would degrade, resulting in an unacceptable user experience. If this commercial only airs twice a week, however, these additional resources might be idle during the rest of the week, consuming additional power without being utilized.

Devising mecahnisms for reducing power consumption and environmental impact through cloud auto-scaling is hard. Auto-scaling must ensure that VMs can be provisioned and booted quickly to meet response time requirements as load changes. If auto-scaling responds to load fluctuations too slowly applications may experience a period of poor response time awaiting the allocation of additional computational resources. One way to mitigate this risk is to maintain an auto-scaling queue containing prebooted and preconfigured VM instances that can be allocated rapidly, as shown in Figure 1.

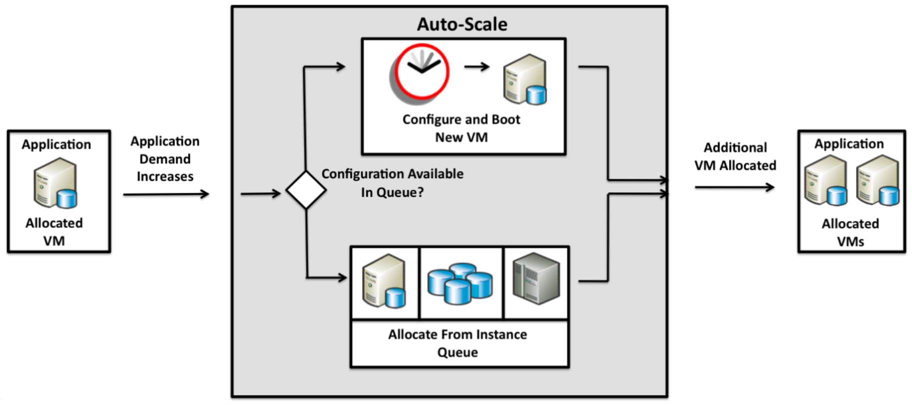

Application
Allocated VM
Application Demand Increases
Auto-Scale
Configure and Boot New VM
Configuration Available In Queue?
Allocate From Instance Queue
Additional VM Allocated
Application
Allocated VMs

Figure 1: Auto-scaling in a Cloud Infrastructure

When a cloud application requests a new VM configuration from the auto-scaling infrastructure, the auto-scaling infrastructure first attempts to fulfill the request with a prebooted VM in the queue. For example, if a VM with Fedora Core 6, JBoss, and

MySQL is requested, the auto-scaling infrastructure will attempt to find a matching VM in the queue. If no match is found, a new VM must be booted and configured to match the allocation request.

Open problem → determining green settings, such as the size and properties of the auto-scaling queue shared by multiple applications with different VM configurations [10]. The chosen configurations must meet the configuration requirements of multiple applications and reduce power consumption without adversely impacting QoS. For example, a web application may request VM instances configured as database, middle-tier Enterprise Java Beans (EJB), or front-end web servers. Determining how to capture and reason about the configurations that comprise the auto-scaling queue is hard due to the large number of configuration options (such as MySQL and SQL Server databases, Ubuntu Linux and Windows operating systems, and Apache HTTP and IIS/Asp.Net web hosts) offered by cloud infrastructure providers.

It is even harder to determine the optimal queue size and types of VM configurations that will ensure VM allocation requests can be serviced quickly enough to meet a required auto-scaling response time limit. Cost optimization is challenging because each configuration placed into the queue can have varying costs based on the hardware resources and software licenses it uses. Energy consumption minimization is also hard since hardware resources can consume different amounts of power.

Solution approach → Auto-scaling queue configuration derivation based on feature models. This paper presents a model-driven engineering (MDE) approach called the *Smart Cloud Optimization for Resource Configuration Handling* (SCORCH). SCORCH captures VM configuration options for a set of cloud applications and derives an optimal set of virtual machine configurations for an auto-scaling queue to provide three green computing contributions:

- An MDE technique for transforming feature model representations of cloud VM configuration options into constraint satisfaction problems (CSPs) [11, 12], where a set of variables and a set of constraints govern the allowed values of the variables.
- An MDE technique for analyzing application configuration requirements, VM power consumption, and operating costs to determine what VM instance configurations an auto-scaling queue should contain to meet an auto-scaling response time guarantee while minimizing power consumption.
- Empirical results from a case study using Amazon's EC2 cloud computing infrastructure (`aws.amazon.com/ec2`) that shows how SCORCH minimizes power consumption and operating cost while ensuring that auto-scaling response time requirements are met.

2 Challenges of Configuring Virtual Machines in Cloud Environments

Reducing unnecessary idle system resources by applying auto-scaling queues can potentially reduce power consumption and resulting CO_2 emissions significantly. QoS demands, diverse configuration requirements, and other challenges, however, make it hard to achieve a greener computing environment. This section describes three key challenges of capturing VM configuration options and using this configuration information to optimize the setup of an auto-scaling queue to minimize power consumption.

2.1 *Challenge 1: Capturing VM Configuration Options and Constraints*

Cloud computing can yield greener computing by reducing power consumption. A cloud application can request VMs with a wide range of configuration options, such as type of processor, OS, and installed middleware, all of which consume different amounts of power. For example, the Amazon EC2 cloud infrastructure supports 5 different types of processors, 6 different memory configuration options, and over 9 different OS types, as well as multiple versions of each OS type [13]. The power consumption of these configurations range from 150 to 610 watts per hour.

The EC2 configuration options cannot be selected arbitrarily and must adhere to myriad configuration rules. For example, a VM running on Fedora Core 6 OS cannot run MS SQL Server. Tracking these numerous configuration options and constraints is hard. Sections 3.1&3.2 describe how SCORCH uses feature models to alleviate the complexity of capturing and reasoning about configuration rules for VM instances.

2.2 *Challenge 2: Selecting VM Configurations to Guarantee Auto-scaling Speed Requirements*

While reducing idle resources results in less power consumption and greener computing environments, cloud computing applications must also meet stringent QoS demands. A key determinant of auto-scaling performance is the types of VM configurations that are kept ready to run. If an application requests a VM configuration and an exact match is available in the auto-scaling queue, the request can be fulfilled nearly instantaneously. If the queue does not have an exact match, it may have a running VM configuration that can be modified to meet the requested configuration faster than provisioning and booting a VM from scratch. For example, a configuration may reside in the queue that has the correct OS but needs to unzip a custom software package, such as a pre-configured Java Tomcat Web Application Server, from a shared filesystem onto the VM. Auto-scaling requests can thus be fulfilled with both exact configuration matches and subset configurations that can be modified faster than provisioning a VM from scratch.

Determining what types of configurations to keep in the auto-scaling queue to ensure that VM allocation requests are serviced fast enough to meet a hard allocation time constraint is hard. For one set of applications, the best strategy may be to fill the queue with a common generic configuration that can be adapted quickly to satisfy requests from each application. For another set of applications, it may be faster to fill the queue with the virtual machine configurations that take the longest to provision from scratch. Numerous strategies and combinations of strategies are possible, making it hard to select configurations to fill the queue that will meet auto-scaling response time requirements. Section 3.3 show how SCORCH captures cloud configuration options and requirements as cloud configuration feature models, transforms these models into a CSP, and creates constraints to ensure that a maximum response time limit on auto-scaling is met.

2.3 *Challenge 3: Optimizing Queue Size and Configurations to Minimize Energy Consumption and Operating Cost*

A further challenge for developers is determining how to configure the auto-scaling queue to minimize the energy consumption and costs required to maintain it. The larger the queue, the greater the energy consumption and operating cost. Moreover, each individual configuration within the queue varies in energy consumption and cost.For example, a “small” Amazon EC2 VM instance running a Linux-based OS consumes 150W and costs $0.085 per hour while a "Quadruple Extra Large" VM instance with Windows consumes 630W and costs $2.88 per hour.

It is hard for developers to manually navigate tradeoffs between energy consumption, operating costs, and auto-scaling response time of different queue sizes and sets of VM configurations. Moreover, there are an exponential number of possible queue sizes and configuration options that complicates deriving the minimal power consumption/operating cost queue configuration that will meet auto-scaling speed requirements. Section 3.3 describes how SCORCH uses CSP objective functions and constraints to derive a queue configuration that minmizes power consumption and operating cost.

3 The Structure and Functionality of SCORCH

This section describes how SCORCH resolves the challenges in Section 2 by using (1) models to capture virtual machine configuration options explicitly, (2) model transformations to convert these models into CSPs, (3) constraint solvers to derive the optimal queue size, and (4) contained VM configuration options to minimize energy consumption and operating cost while meeting auto-scaling response time requirements.

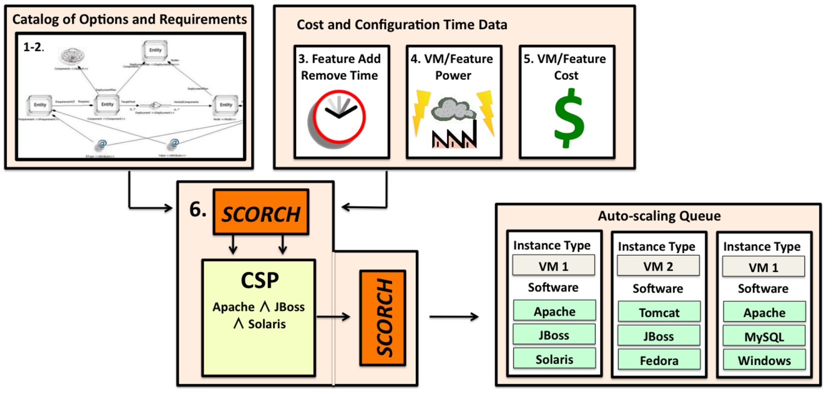


Figure 2: SCORCH Model-Driven Process

The SCORCH MDE process is shown in Figure 2 and described below:

1. Developers use a SCORCH *cloud configuration model* to construct a catalog of configuration options that are available to VM instances.

2. Each application considered in the auto-scaling queue configuration optimization provides a *configuration demand model* that specifies the configuration for each type of virtual machine instance the application will request during its execution lifecycle.

3. Developers provide a *configuration adaptation time model* that specifies the time required to add/remove a feature from a configuration.

4. Developers provide an *energy model* that specifies the power consumption required to run a VM configuration with each feature present in the SCORCH cloud configuration model.

5. Developers provide a *cost model* that specifies the cost to run a VM configuration with each feature present in the SCORCH cloud configuration model.

6. The cloud configuration model, configuration demand models, and load estimation model are transformed into a CSP and a constraint solver is used to derive the optimal auto-scaling queue setup.

The remainder of this section describes the structure and functionality of each model defined and used by SCORCH.

3.1 SCORCH Cloud Configuration Models

A key consideration in SCORCH is modeling the catalog of VM configuration options. Amazon EC2 offers many different options, such as Linux vs. Windows operating systems, SQL Server vs. MySQL databases, and Apache HTTP vs. IIS/Asp.Net webhosts. This model provides developers with a blueprint for constructing a request for a VM instance configuration and checking its correctness. The queue configuration optimization process also uses this model to ensure that valid configurations are chosen to fill the queue.

To manage the complexity of representing VM instance configuration options, SCORCH uses *feature models* [12], which describe commonality and variability in a configurable software platform via an abstraction called a *feature*. Features can describe both high-level functional variations in the software, *e.g.*, whether or not the underlying software can load balance HTTP requests. A feature can also represent implementation-specific details, *e.g.*, whether or not Ubuntu 9.10 or Fedora is used.

Feature models use a tree structure to define the relationships between the various features and encode configuration rules into the model, *e.g.*, a VM configuration can include only a single operating system, such as Ubuntu 9.10 or Fedora. Some features may require other features to be present to function, *e.g.*, the `JBOSS v6` feature cannot be chosen without also selecting the `JBOSS` feature.

A configuration of the software platform is defined by a selection of features from the feature model. The most basic rule of configuration correctness is that every selected feature must also have its parent feature selected. This rule also implies that every correct feature selection must include the root feature. Moreover, the feature selection must adhere to the constraints on the parent-child relationships encoded into the feature model.

Developers use the SCORCH cloud configuration model to express the available configuration options for VM instances as a feature model. The configuration adaption time model's information is captured as attributes of the features in the SCORCH cloud configuration model. Each feature can be annotated with an integer attribute that specifies the time in milliseconds to add/remove the given feature from a configuration.

The energy model and cost model are also captured using attributes in the SCORCH cloud configuration model. Each feature impacting the energy consumption or operating cost of a configuration is annotated with an energy attribute that specifies the energy consumption per hour and cost attribute that specifies the operating cost per hour to have a booted VM configuration in the queue with that feature. For example, these attributes can be used to model the cost of the "Small" vs. "Quadruple Extra Large" computing node size features of an Amazon EC2 VM configuration.

3.2 SCORCH Configuration Demand Models

Applications are auto-scaled at runtime by dynamically requesting and releasing VM instances. When a new VM instance is requested, the desired configuration for the instance is provided. SCORCH requires each application to provide a model of the VM instance configurations that it will request over its lifetime.

Developers construct SCORCH configuration demand models to dictate what VM configurations an application will request. The configuration demand models use a textual domain-specific language to describe each configuration requested as a selection of features from the SCORCH cloud configuration model.

3.3 Runtime Model Transformation to CSP and Optimization

Using feature models to capture VM configuration options allows the use of constraint solvers to select a group of features to optimize an objective function. In the context of SCORCH, the cloud configuration model and configuration demand models are converted into a CSP where a solution is a valid set of configurations for the VM instances in the auto-scaling queue. The objective function of the CSP attempts to derive a mix of configurations that minimizes the energy consumption and cost of maintaining the queue while ensuring that any hard constraints on the time to fulfill auto-scaling requests are met.

The conversion of feature selection problems into CSPs has been described in prior work [14, 15]. Feature configuration problems are converted into CSPs where the selection state of each feature is represented as a variable with domain {0,1}. The constraints are designed so that a valid labeling of these variables yields a valid feature selection from the feature model.

A CSP for a feature selection problem can be described as a 3-tuple:

$$P =< F, C, \gamma >$$

where:

• F is a set of variables describing the selection state of each feature. For each feature, $f_i \in F$, if the feature is selected in the derived configuration, then $f_i = 1$. If the i^{th} feature is not selected, then $f_i = 0$.

• C captures the rules from the feature model as constraints on the variables in F. For example, if the i^{th} feature requires the j^{th} feature, C would include a constraint: $(f_i = 1) \Rightarrow (f_j = 1)$.

• γ is an optional objective function that should be maximized or minimized by the derived configuration.

Building a CSP to derive a set of configurations for an auto-scaling queue uses a similar methodology. Rather than deriving a single valid configuration, however, SCORCH tries to simultaneously derive both the size of the auto-scaling queue and a configuration for each position in the auto-scaling queue. If SCORCH derives a size for the queue of K, therefore, K different feature configurations will be derived for the K VM instances that need to fill the queue.

The CSP for a SCORCH queue configuration optimization process can be described formally as the 8-tuple

$$P =< S, Q, C, D, E, L, T, M, \gamma >$$

, where:

• S is the auto-scaling queue size, which represents the number of prebooted VM instances available in the queue. This variable is derived automatically by SCORCH.

• Q is a set of sets that describes the selection state of each VM instance configuration in the queue. The size of Q is Z if there are Z distinct types of configurations specified in the configuration demand models. Each set of variables, $Q_i \in Q$, describes the selection state of features for one VM instance in the queue. For each variable, $q_{ij} \in Q_i$, if $q_{ij} = 1$ in a derived configuration, it indicates that the j^{th} feature is selected by the i^{th} VM instance configuration.

• C captures the rules from the feature model as constraints on the variables in all sets $Q_i \in Q$. For example, if the kth feature requires the j^{th} feature, C would include a constraint: $\forall Q_i \in Q,\ (q_{ik} = 1) \Rightarrow (q_{ij} = 1)$.

• D contains the set of configuration demand models contributed by the applications. Each demand model $D_i \in D$ represents a complete set of selection states for the features in the feature model. If the j^{th} feature is requested by the i^{th} demand model, then $d_i j \in D_i, d_i j = 1$. The demand models can be augmented with expected load per configuration, which is a focus of future work.

• E is the cost model that specifies the energy consumption resulting from including the feature in a running VM instance configuration in the auto-scaling queue. For each configuration $D_i \in D$ a variable $E_i \in E$ specifies the energy consumption of that feature. These values are derived from annotations in the SCORCH cloud configuration model.

• L is the cost model that specifies the cost to include the feature in a running VM instance configuration in the auto-scaling queue. For each configuration $D_i \in D$ a variable $L_i \in L$ specifies the cost of that feature. These values are derived from annotations in the SCORCH cloud configuration model.

• T is the configuration time model that defines how much time is needed to add/-remove a feature from a configuration. The configuration time model is expressed as a set of positive decimal coefficients, where $t_i \in T$ is the time required to add/remove the i^{th} feature from a configuration. These values are derived from the annotations in the SCORCH cloud configuration model.

• γ is the cost minimization objective function that is described in terms of the variables in D, Q, and L.

• M is the maximum allowable response time to fulfill a request to allocate a VM with any requested configuration from the demand models to an application.

3.4 Response Time Constraints and CSP Objective Function

SCORCH defines an objective function to attempt to minimize the cost of maintaining the auto-scaling queue, given a CSP to derive configurations to fill the queue. Moreover, we can define constraints to ensure that a maximum response time bound is adhered to by the chosen VM queue configuration mix and queue size that is derived.

We describe the expected response time, Rt_x, to fulfill a request D_x from the configuration demand model as:

$$Rt_x = \min(CT_0 \ldots CT_n,\ boot(D_x)) \tag{1}$$

$$CT_i = \begin{cases} \forall q_{ij} \in Q_i,\ q_{ij} = d_{xj} & 0\ (a), \\ \exists q_{ij} \in Q_i,\ q_{ij}! = d_{xj} & \sum t_j(|q_{ij} - d_{xj}|)\ (b) \end{cases} \tag{2}$$

where:

- Rt_x is the expected response time to fulfill the request.
- n is the total number of features in the SCORCH cloud configuration model
- CT_i is the expected time to fulfill the request if the i^{th} VM configuration in the queue was used to fulfill it.
- $boot(D_x)$ is the time to boot a new VM instance to satisfy D_x and not use the queue to fulfill it.

The expected response time, Rt_x is equal to the fastest time available to fulfill the request, which will either be the time to use a VM instance in the queue CT_i or to boot a completely new VM instance to fulfill the request $boot(D_x)$. The time to fulfill the request is zero (or some known constant time) if a configuration exists in the queue that exactly matches request (a). The time to fulfill the request with that configuration is equal to the time needed to modify the configuration to match the requested configuration D_x if a given VM configuration is not an exact match (b). For each feature q_{ij} in the configuration that does not match what is requested in the configuration, t_j is the time incurred to add/remove the feature. Across the Z distinct types of configuration requests specified in the configuration demand models we can therefore limit the maximum allowable response time with the constraint:

$$\forall D_x \in D,\ M \geq Rt_x \tag{3}$$

With the maximum response time constraint in place, the SCORCH model-to-CSP transformation process then defines the objective function to minimize. For each VM instance configuration, Q_i, in the queue we define its energy consumption as:

$$Energy(Q_i) = \sum_{j=0}^{n} q_{ij}E_j$$

. The overall energy consumption minimization objective function, ε, is defined as the minimization of the variable $Energy$, where:

$$\varepsilon = Energy = Energy(Q_0) + Energy(Q_1) + \cdots + Energy(Q_k)$$

.

Similarly, the cost of each VM instance is defined as:

$$Cost(Q_i) = \sum_{j=0}^{n} q_{ij}L_j$$

. The overall cost minimization objective function, γ, is defined as the minimization of the variable $Cost$, where:

$$\gamma = Cost = Cost(Q_0) + Cost(Q_1) + \cdots + Cost(Q_k)$$

.

The final piece of the CSP is defining the constraints attached to the queue size variable S. We define S as the number of virtual machine instance configurations that have at least one feature selected:

$$S_i = \begin{cases} \forall q_{ij} \in Q_i,\ q_{ij} = 0 & 0, \\ \exists q_{ij} \in Q_i,\ q_{ij} = 1 & 1 \end{cases} \tag{4}$$

$$S = \sum_{i=0}^{Z} S_i$$

Once the CSP is constructed, a standard constraint solver, such as the Java Choco constraint solver (`choco.sourceforge.net`), can be used to derive a solution. Section 4 presents empirical results from applying SCORCH with Java Choco to a case study of an ecommerce application running on Amazon's EC2 cloud computing infrastructure.

4 Empirical Results

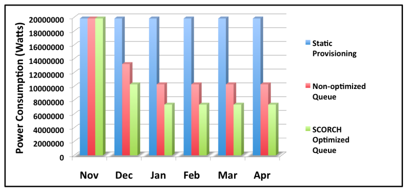
This section presents a comparison of SCORCH with two other approaches for provisioning VMs to ensure that load fluctuations can be met without degradation of QoS. We compare the energy efficiency and cost effectiveness of each approach when provisioning an infrastructure that supports a set of ecommerce applications. We selected ecommerce applications due to the high fluctuations in workload that occur due to the varying seasonal shopping habits of users. To compare the energy efficiency and cost effectiveness of these approaches, we chose the pricing model and available VM instance types associated with Amazon EC2.

We investigated three-tiered ecommerce applications consisting of web front end, middleware, and database layers. The applications required 10 different distinct VM configurations. For example, one VM required JBOSS, MySql, and IIS/Asp.Net while another required Tomcat, HSQL, and Apache HTTP. These applications also utilize a variety of computing instance types from EC2, such as high-memory, high-CPU, and standard instances.

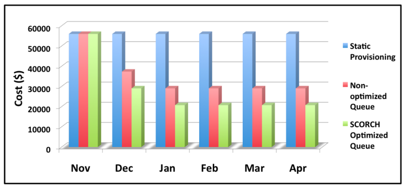
To model the traffic fluctuations of ecommerce sites accurately we extracted traffic information from Alexa (`www.alexa.com`) for newegg.com (`newegg.com`), which is an extremely popular online retailer. Traffic data for this retailer showed a spike of three times the normal traffic during the November-December holiday season. During this period of high load, the site required 54 VM instances. Using the pricing model provided by Amazon EC2, each server requires 515W of power and costs \$1.44 an hour to support the heightened demand (`aws.amazon.com/economics`).

4.1 Experiment: VM Provisioning Techniques

Static provisioning. The first approach provisions a computing infrastructure equipped to handle worst-case demand at all times. In this approach, all 54 servers run continuously to maintain response time. This technique is similar to computing environments that permit no auto-scaling. Since the infrastructure can always support the worst-case load, we refer to this technique as *static provisioning*.



(a) Monthly Power Consumption



(b) Monthly Cost

Figure 3: Monthly Power Consumption & Cost

Non-optimized auto-scaling queue. The second approach augments the auto-scaling capabilities of a cloud computing environment with an auto-scaling queue. In this approach, auto-scaling is used to adapt the number of resources to meet the current load that the application is experiencing. Since additional resources can be allocated as demand increases, we need not run all 54 servers continuously. Instead, an auto-scaling queue with a VM instance for each of ten different application configurations must be allocated on demand. We refer to this technique as *non-optimized auto-scaling queue* since the auto-scaling queue is not optimized.

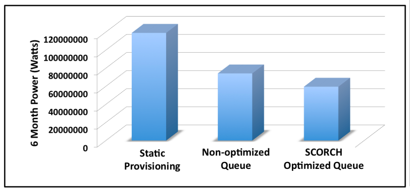
SCORCH. The third approach uses SCORCH to minimize the number of VM instances needed in the auto-scaling queue, while ensuring that response time is met. By optimizing the auto-scaling queue with SCORCH, the size of the queue can be reduced by 80% to two VM instances.

4.2 Power Consumption & Cost Comparison of Techniques

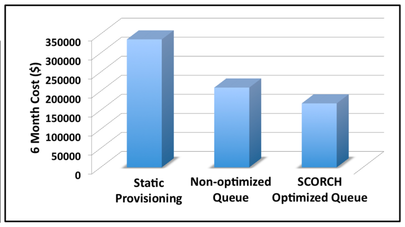
The maximum load for the 6 month period occurred in November and required 54 VM instances to support the increased demand, decreasing to 26 servers in December and finally 18 servers for the final four months. The monthly energy consumption and operational costs of applying each response time minimization technique can be seen in Figure 3a and 3b respectively.

Since the maximum demand of the ecommerce applications required 54 VM instances, the static provisioning technique consumed the most power and was the most expensive, with 54 VM instances prebooted and run continuously. The non-optimized auto-scaling queue only required ten pre-booted VM instances and therefore reduced power consumption and cost. Applying SCORCH yielded the most energy efficient and lowest cost infrastructure by requiring only two VM instances in the auto-scaling queue.

Figures 4a and 4b compares the total power consumption and operating cost of applying each of the VM provisioning techniques for a six month period. The non-optimized auto-scaling queue and SCORCH techniques reduced the power requirements and price of utilizing an auto-scaling queue to maintain response time in comparison to the static provisioning technique. Figure 5a compares the savings of using a non-optimized auto-scaling queue versus an auto-scaling queue generated with SCORCH. While both techniques reduced cost by more than 35%, deriving an auto-scaling queue configuration with SCORCH yielded a 50% reduction of cost com-

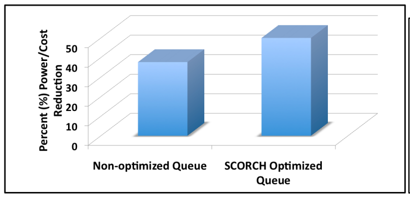


(a) Total Power Consumption

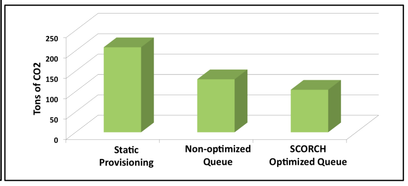


(b) Total Cost

Figure 4: Monthly Power Consumption & Cost



(a) Power Consumption/Cost Percent Reduction



(b) CO2 Emissions

Figure 5: Environmental Impact of Techniques

pared to utilizing the static provisioning technique. This result reduced costs by over $165,000 for supporting the ecommerce applications for 6 months.

More importantly than reducing cost, however, applying SCORCH also reduced CO_2 emissions by 50%, as shown in Figure 5b. According to recent studies, a power plant using pulverized coal as its power source emits 1.753 pounds of CO_2 per each kilowatt hour of power produced [2]. Not using an auto-scaling queue therefore results in an emission of 208.5 tons of CO_2 per year, as shown in Figure 5b. Applying the SCORCH optimized auto-scaling queue, however, cuts emissions by 50% resulting in an emission reduction of 104.25 tons per year.

5 Related Work

This section compares SCORCH with related work.

VM forking handles increased workloads by replicating VM instances onto new hosts in negligible time, while maintaining the configuration options and state of the original VM instance. Cavilla et al. [16] describe SnowFlock, which uses virtual machine forking to generate replicas that run on hundreds of other hosts in a less than a second. This replication method maintains both the configuration and state of the cloned machine. Since SnowFlock was designed to instantiate replicas on multiple physical machines, it is ideal for handling increased workload in a cloud computing environment where large amounts of additional hardware is available.

SnowFlock is effective for cloning VM instances so that the new instances have the same configuration and state of the original instance. As a result, the configuration and boot time of a VM instance replica can be almost entirely bypassed. This technique, however, requires that at least a single virtual machine instance matching the configuration requirements of the requesting application is booted. In contrast, SCORCH uses prebooted VM instances that are more likely to match the configuration requirements of arriving applications.

Automated feature derivation. To maintain the service-level agreements provided by cloud computing environments, it is critical that techniques for deriving VM instance configurations are automated since manual techniques cannot support the dynamic scalability that makes cloud computing environments attractive. Many techniques [17, 18, 19, 20] exist to automatically derive feature sets from feature models. These techniques convert feature models to CSPs that can be solved using commercial CSP solvers. By representing the configuration options of VM instances as feature models, these techniques can be applied to yield feature sets that meet the configuration requirements of an application. Existing techniques, however, focus on meeting configuration requirements of one application at a time. These techniques could therefore be effective for determining an exact configuration match for a single application. In contrast, SCORCH analyzes CSP representations of feature models to determine feature sets that satisfy some or all of feature requirements of multiple applications.

6 Concluding Remarks

Auto-scaling cloud computing environments helps minimize response time during periods of high demand, while reducing cost during periods of light demand. The time to boot and configure additional VM instances to support applications during periods of high demand, however, can negatively impact response time. This paper describes how the *Smart Cloud Optimization of Resource Configuration Handling* (SCORCH) MDE tool uses feature models to (1) represent the configuration requirements of multiple software applications and the power consumption/operational costs of utilizing different VM configurations, (2) transform these representations into CSP problems, and (3) analyze them to determine a set of VM instances that maximizes auto-scaling queue hit rate. These VM instances are then placed in an auto-scaling queue so that response time requirements are met while minimizing power consumption and operational cost.

The following are lessons learned from using SCORCH to construct auto-scaling queues that create greener computing environments by reducing emissions resulting from superfluous idle resources:

• **Auto-scaling queue optimization effects power consumption and operating cost.** Using an optimized auto-scaling queue greatly reduces the total power consumption and operational cost compared to using a statically provisioned queue or non-optimized auto-scaling queue. SCORCH reduced power consumption and operating cost by 50% or better.

• **Dynamic pricing options should be investigated.** Cloud infrastructures may change the price of procuring VM instances based on current overall cloud demand at a given moment. We are therefore extending SCORCH to incorporate a monitoring system that considers such price drops when appropriate.

• **Predictive load analysis should be integrated.** The workload of a demand model can effect application resource requirements drastically. We are therefore extending SCORCH to use predictive load analysis so auto-scaling queues can cater to specific application workload characteristics.

SCORCH is part of the ASCENT Design Studio and is available in open-source format from `code.google.com/p/ascent-design-studio`.

References

[1] Computer center powernap plan could save 75 percent of data center energy, `http://www.sciencedaily.com/releases/2009/03/090305164353.htm`, accessed October 20, 2010 (2009).

[2] E. Rubin, A. Rao, C. Chen, Comparative assessments of fossil fuel power plants with CO2 capture and storage, in: Proceedings of 7th International Conference on Greenhouse Gas Control Technologies, Vol. 1, 2005, pp. 285–294.

[3] C. Cassar, Electric power monthly, `http://www.eia.doe.gov/cneaf/electricity/epm/epm_sum.html`, accessed October 20, 2010.

[4] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Dang, K. Pentikousis, Energy-efficient cloud computing, The Computer Journal 53 (7) (2010) 1045.

[5] L. Liu, H. Wang, X. Liu, X. Jin, W. He, Q. Wang, Y. Chen, GreenCloud: a new architecture for green data center, in: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session, ACM, 2009, pp. 29–38.

[6] A. Bateman, M. Wood, Cloud computing, Bioinformatics 25 (12) (2009) 1475.

[7] A. Beloglazov, R. Buyya, Energy efficient allocation of virtual machines in cloud data centers, in: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, IEEE, 2010, pp. 577–578.

[8] R. Buyya, A. Beloglazov, J. Abawajy, Energy-Efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges, in: Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010), Las Vegas, USA, July 12, Vol. 15, 2010.

[9] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The eucalyptus open-source cloud-computing system, in: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00, IEEE Computer Society, 2009, pp. 124–131.

[10] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Obbink, K. Pohl, Variability issues in software product lines, Lecture Notes in Computer Science (2002) 13–21.

[11] V. Kumar, Algorithms for constraint-satisfaction problems: A survey, AI magazine 13 (1) (1992) 32.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-oriented domain analysis (FODA) feasibility study (1990).

[13] S. Hazelhurst, Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud, in: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, ACM New York, NY, USA, 2008, pp. 94–103.

[14] D. Benavides, P. Trinidad, A. Ruiz-Cortes, Automated Reasoning on Feature Models, in: Proceedings of the 17th Conference on Advanced Information Systems Engineering, ACM/IFIP/USENIX, Porto, Portugal, 2005.

[15] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, A. Ruiz-Cortez, Automated Diagnosis of Product-line Configuration Errors in Feature Models, in: Proceedings of the Software Product Lines Conference (SPLC), Limerick, Ireland, 2008.

[16] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. de Lara, M. Brudno, M. Satyanarayanan, SnowFlock: rapid virtual machine cloning for cloud computing, in: Proceedings of the fourth ACM european conference on Computer systems, ACM, 2009, pp. 1–12.

[17] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Vol. 3520, Springer, 2005, pp. 491–503.

[18] J. White, B. Dougherty, D. Schmidt, Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening, The Journal of Systems & Software 82 (8) (2009) 1268–1284.

[19] J. White, D. Benavides, B. Dougherty, D. Schmidt, Automated Reasoning for Multi-step Configuration Problems, in: Proceedings of the Software Product Lines Conference (SPLC), San Francisco, USA, 2009.

[20] J. White, D. Schmidt, D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated diagnosis of product-line configuration errors in feature models, in: Proceedings of the Software Product Lines Conference (SPLC), Citeseer, 2008, pp. 225–234.