CloudAlloc: A Monitoring and Reservation System for Compute Clusters

Enrico Iori[1] Alkis Simitsis[2] Themis Palpanas[1] Kevin Wilkinson[2] Stavros Harizopoulos[3]

[1]University of Trento
enrico.iori.tv@gmail.com,
themis@disi.unitn.eu

[2]HP Labs
alkis@hp.com,
kevin.wilkinson@hp.com

[3]Nou Data
stavros@noudata.com

ABSTRACT

Cloud computing has emerged as a promising environment capable of providing flexibility, scalability, elasticity, fail-over mechanisms, high availability, and other important features to applications. Compute clusters are relatively easy to create and use, but tools to effectively share cluster resources are lacking. *CloudAlloc* addresses this problem and schedules workloads to cluster resources using allocation algorithms. It also monitors resource utilization and thus, provides accountability for actual usage. *CloudAlloc* is a lightweight, flexible, easy-to-use tool for cluster resource allocation that has also proved useful as a research platform. We demonstrate its features and also discuss its allocation algorithms that minimize power usage. *CloudAlloc* was implemented and is in use at HP Labs.

Categories and Subject Descriptors

H.2.m [**Information Systems Applications**]: Miscellaneous

Keywords

Cloud Computing, Reservation System, Monitoring System

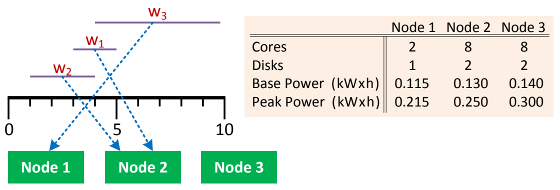
1. INTRODUCTION

Open source toolkits and cloud service providers have made it easy to create and use large compute clusters. However, efficient sharing of such clusters is an open problem for both private clouds and public clouds. For private clouds, sharing is often done by exclusive reservation of nodes which results in low resource utilization and overprovisioning. For public clouds, applications typically run on multiple virtual machines that are assigned to nodes by an algorithm that is not exposed to applications. This makes cloud-based applications dependent on the cloud provider for service level guarantees, which eliminates an important control point for an application provider to differentiate its services to users. For example, the objectives of the cloud service provider, e.g., maximize utilization, may be at odds with the needs of the application provider, e.g., maximize throughput.

In our work, we address the problem of cluster resource sharing. Given a set of workloads over time, the task is to assign the workloads to cluster nodes in a way that satisfies an objective (see Figure 1). In our model, a workload request comprises a set of resources (e.g., 20 cores at 30 percent utilization, 2GB per core, 5 IO/sec per core), a duration (e.g., 4 hours), and optionally, a start time. If the request is satisfiable, the reservation is made. Some possible objectives for the assignment include maximize utilization of each node, minimize completion time, minimize power usage, and so on. In a static environment, where the applications are fixed and the workload is known, a single resource allocation algorithm with fixed objectives may be highly tuned and work well. However, in a dynamic environment such as an R&D lab or a public, shared cluster, the ability to vary the allocation algorithm and objectives is desirable. For example, during the day, when energy costs are high, the objective may be to minimize power usage while in the evening, the objective may be altered to minimize execution time.

We present *CloudAlloc*, a lightweight tool for flexibly allocating resources of a compute cluster to workloads. *CloudAlloc* is intended as a tool for use in R&D environments, but it may also serve as a research vehicle for workload management. Along those lines, we demonstrate the capabilities of *CloudAlloc* through five scenarios. *(i)* In steady state operation, *CloudAlloc* monitors resource usage on each cluster node. A dashboard provides a display of the past resource usage and future reservations at different levels of granularity. *(ii)* When a new workload is submitted to reserve resources in the future, *CloudAlloc* passes that request, the current cluster state, and future reservations to an allocation algorithm to determine an assignment. If the request is satisfiable, the resources are reserved. *(iii) CloudAlloc* allows system administrators to vary the resource allocation algorithm. Our demonstration includes two resource allocation algorithms that minimize cluster power usage, but *CloudAlloc* has a plug-in architecture so additional algorithms with other objectives can be added. *(iv)* When a workload runs, *CloudAlloc* records the actual resource usage. In this way, users can refine their resource estimates and administrators may correlate workload requests against actual usage. *(v) CloudAlloc* handles cluster reconfiguration so that when nodes are added or fail, the resource allocator is reexecuted to reassign workloads for the new configuration.

Next, we discuss background work, present the system architecture, and explain the scenarios of our demonstration.



	Node 1	Node 2	Node 3
Cores	2	8	8
Disks	1	2	2
Base Power (kWxh)	0.115	0.130	0.140
Peak Power (kWxh)	0.215	0.250	0.300

Figure 1: Example workload assignment

2. BACKGROUND

State of the art. Several compute cluster frameworks provide a scheduling component for assigning jobs to nodes (e.g., Condor [7], Platform LSF [2], SLURM [6]). However, jobs are scheduled as they arrive, rather than advance reservations and the algorithms are fixed. Nimrod/G is a tool focused on computational experiments [1]. Power consumption and service provisioning in a cloud environment [3, 8, 9] has gained popularity. *CloudAlloc* is different in more than one way since other approaches *(i)* assume a fixed cost for computing nodes, which does not accurately model the energy cost in our setting, *(ii)* focus on particular requirements of data transfer and storage and thus lack our generality, *(iii)* do not consider the start/end time requirements for workloads. For example, Kairos analyzes workloads and decides the placement of tasks to physical machines in the cloud [4], but is geared towards relational database workloads which have some specific characteristics. A more detailed comparison with the related work can be found in [5].

Theoretical underpinnings. *CloudAlloc* offers two algorithms for solving the workload allocation problem where low power usage is the objective. We assume a heterogeneous cluster, i.e., the nodes need not to be identical.

Optimal solution. We formulate the problem as a linear programming problem, where we minimize an objective function subject to a number of constraints. Specifically, we need to satisfy the resource requirements of the workloads (e.g., CPU, disk usage, network bandwidth) such that the produced solution is feasible and semantically correct (e.g., each node is not assigned more workloads than what it can accommodate, each workload is serviced exactly once). Our objective function is formed as the total power cost (both base and peak power) for all the nodes in the cloud given some workload. The objective function comprises two parts related to the base (idle) power of each node and the percentage of peak power consumed by each workload on each node. Given q nodes and s workloads, it is as follows:

$$\sum_{j=1}^{q}\sum_{i=1}^{s}\left(\frac{\omega_j^b}{u_j^w}\cdot v_j+\frac{c_i\cdot\omega_j^p}{c_j^t}\cdot(e_i-s_i)\cdot x_{ij}\right) \qquad (1)$$

ω_j^b and ω_j^p are the base and peak power of node n_j, u_j^w is the number of workloads executed in node n_j, c_i is the number of cores needed by workload i, and c_j^t is the total number of cores of n_j. The v_j variable expresses the duration related with all the workloads assigned to n_j, x_{ij} is a binary variable that is one if w_i is assigned to n_j and zero otherwise, and $(e_i - s_i)$ is the duration (start and end times) of workload w_i. This problem is solved using linear programming, producing the optimal solution. Note that c can be a fraction; e.g, c_i=0.5 means that either a single core will be 50% utilized or 2 cores will be each 25% utilized. The latest generation of CPUs come increasingly close to a linear model for power consumption (vs. utilization) once a fixed component is taken into account (idle power). Thus, we average the CPU utilization for the entire task duration. Also, with c we assign different CPU utilization levels to different workloads (and thus, different CPU intensity). However, we do not attempt to model any system/platform-level interference (e.g., CPU cache misses, context-switch overheads).

Greedy solution. Our greedy approach is an approximate, yet fast, solution to the workload allocation problem. The algorithm starts by sorting the nodes according to the electrical power consumption per core, $[\omega_j^b + (\omega_j^p \cdot c_j^t)]/c_j^t$. Then, it sorts the workloads according to their start time. The list of workloads now is such that the first workload is the one having a start time that is earlier or equal to any other workload. Next, we cycle over all the workloads and, before trying to assign them to some node, we check whether some previously started workload has finished its execution. This can be checked by comparing the start time of the current workload against the end time of all the workloads that have already started. Each workload that has finished its execution (if any) releases the occupied resources. This means that these resources are available for reuse. Finally, we assign the current workload to some nodes. Hence, we start cycling over all the nodes and we assign the workload to the first one that has enough resources to accommodate it.

Example. Consider three nodes n_1, n_2, n_3 and two workloads, w_1 and w_2, both requiring 4 cores and 1 disk, as shown in Figure 1. w_1 starts at time 3 and finishes at 5, while w_2 lasts from 1 to 4. The greedy sorts the nodes according to the power consumption per core, resulting in n_2 being first (with power consumption 0.26625), n_1 second (0.2725), and n_3 third (0.3175). In the list of workloads (sorted by starting time), the first one is w_2 and the second is w_1. We start with w_2 and n_2. Since n_2 has the resources needed to execute w_2, we assign it to that node. Now, the available resources of n_2 are 4 cores and 1 disk. Next, we consider w_1. There are no previously started workloads that have finished their execution, so we do not release any resources. Since n_2 is able to execute w_1, we assign w_1 to n_2, and the algorithm terminates since all workloads were assigned. If a third workload, w_3 arrives at 4 needing 2 cores, it would be assigned to n_2.

Evaluation. Both algorithms have been tested for correctness over a wide range of problem sizes. The greedy is always close to the optimal, especially, as the number of workloads increases. (For a large number of workloads, the greedy identifies more easily high-quality –i.e., low cost– placements, while taking some sub-optimal decisions does not affect much the final solution.) In terms of response time, the optimal scales almost linearly with the number of nodes, but scales less gracefully with the number of workloads. For example, for 30 nodes the optimal needs 1s, 15s, 60s for assigning 10, 30, 40 workloads; for assigning 15 workloads to 50, 75, 100 nodes, it needs 10s, 20s, 30s, respectively. For the same tasks, the greedy needs less than 100msec. The greedy, due to its fast performance and its near-optimal solutions, is the default algorithm for *CloudAlloc*.

3. ARCHITECTURE

The architecture of *CloudAlloc* is depicted in Figure 2. Logically, the architecture comprises n+1 nodes, the n cluster nodes (a.k.a. worker nodes) that are reserved by users plus a master node for reservation and monitoring tasks. In practice, the master node could also serve as a worker node if necessary. We assume that the nodes are shared-nothing and run the same variant of Linux. But, they may be het-

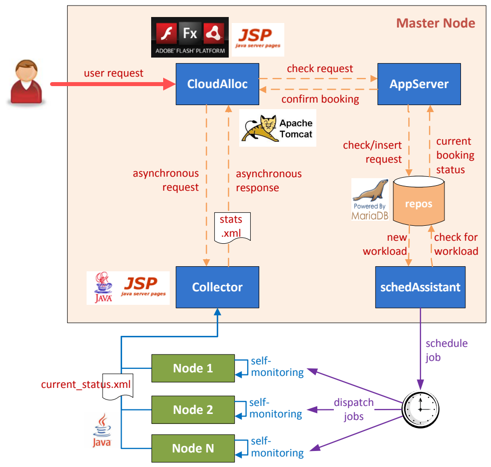


Figure 2: *CloudAlloc* architecture

erogeneous. We model the network as a simple LAN with uniform latency between all nodes. The architecture provides three primary functions and we describe these in turn.

Monitor Function. *CloudAlloc* monitors the cluster at a continuous pace leaving only a light footprint. On each worker node we obtain system statistics using Hyperic's System Information Gatherer (SIGAR), a cross-platform API for collecting software inventory data[1]. Currently we monitor system memory, cpu, disk, and network utilization, but additional statistics may be monitored as needed. We gather these statistics every second and send them as an xml file (current_status.xml in Figure 2) to the master node. For this functionality a lightweight java jar file runs in the background on each node monitored leaving a negligible footprint on memory and cpu utilization. On the master node, every second, a daemon collects statistics from all nodes into one XML file (stats.xml in Figure 2). Nodes may be added to or dropped from the cluster at any time and *CloudAlloc* will incorporate the changes into the reservation system. *CloudAlloc* uses JSP technology to read the stats.xml file and update the *CloudAlloc* dashboard with the appropriate cluster utilization numbers. This data is also stored in the repository (repos in Figure 2) and is used for various purposes.

Reservation Function. *CloudAlloc* orchestrates the workload reservations based on the desired objective function. It is responsible for accommodating workload requests and reserving appropriate time slots on machines for executing all outstanding workloads. It supports a generic API such that new booking algorithms are plug-ins to the system. Currently, *CloudAlloc* supports the two algorithms discussed in Section 2, namely the optimal and approximate solutions. For the optimal solution we used Gurobi[2] Optimizer v.4.0, and for the approximate we used standard Java libraries. All reservations are stored in a database repository. We currently use MariaDB to host our repository. We store information for users (userID, role, etc.), bookings (userID, purpose, etc.), usage (bookingID, machineID, time info, requirements like cpu/memory/diskIO) and so on. It is worth noting that a workload request may be flexible, in which case its reservation may be moved so that *CloudAlloc* may accommodate other urgent workload requests. The GUI for

[1]http://www.hyperic.com/products/sigar

[2]Gurobi (http://www.gurobi.com) is a solver for linear programming, quadratic programming, and mixed-integer programming.

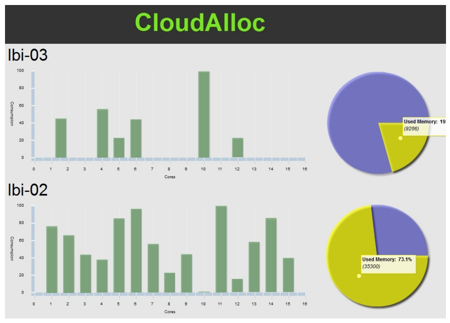


Figure 3: Cluster level monitoring

the monitor, the reservation module, and the administrative features are deployed as web applications in Apache Tomcat and have been implemented using Adobe Flex.

Execution Function. *CloudAlloc* provides a scheduling assistant for executing workloads. Each workload (e.g., queries to be run in a DBMS like Vertica, Hadoop jobs, custom made Unix shell scripts) may have an associated script to initiate execution and that is stored in our repository. The script is parameterized by the number of nodes it needs for execution. A daemon process, schedAssistant, probes the repository for newly registered workloads every 1 minute (a tunable parameter). For each such a script, schedAssistant generates an appropriate command (e.g., composed by script name/path, start time) for the OS scheduler (e.g., cron for Linux). An event is logged in the repository when a workload script starts and ends. That allows us to retrospectively compare the projected and actual resource usage and thus to evaluate the user requirements, the actual system utilization, the accuracy of the process, and to improve our reservation strategy having a better understanding of how each node behaves under load. Clearly, there are several things to be added to our scheduling model (e.g., error handling) and we are working on these for future releases.

4. DEMONSTRATION

Our demonstration testbed will comprise a master node connected to a cluster of worker nodes deployed as virtual machines on Amazon EC2. In actual practice, we would use a cluster of physical nodes. Our use of virtual nodes is just a convenience for the demo, but it also allows us to point out that objectives-based resource allocation of virtual machines running on physical clusters is an open research area.

Our CloudAlloc demonstration will serve two purposes. The first is to show the features of the system. We will work through the five scenarios discussed in Section 1 and solicit feedback on alternative design choices. The second purpose is to present our resource allocation algorithms that minimize power usage and to discuss other possible objectives for resource allocation and alternative power metrics.

Scenario 1: *CloudAlloc* offers monitoring capability at two levels: cluster level (Figure 3) and node level (Figure 4). Cluster monitoring shows the actual utilization of all cluster nodes at any given time. One may choose which nodes to see and what statistics to monitor. Figure 3 shows a fragment of the cluster level monitoring page (only cpu and memory utilization are chosen here). Node level monitoring provides detailed resource usage for a single node (e.g., lbi-02 in Figure 4). It is offered at the booking reservation pages to assist users to make their choices. Similar cluster

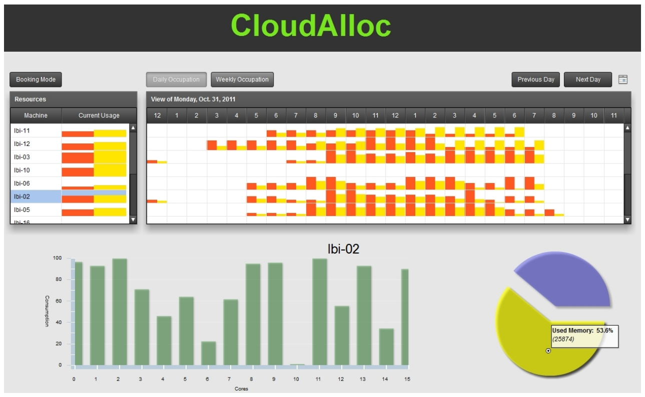
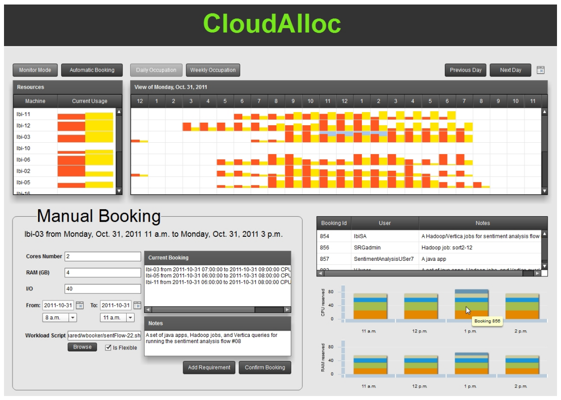


Figure 4: Node level monitoring



lbi-03 from Monday, Oct. 31, 2011 11 a.m. to Monday, Oct. 31, 2011 3 p.m.

Cores Number: 2
RAM (GB): 4
I/O: 40
From: 2011-10-31 8 a.m. To: 2011-10-31 11 a.m.
Workload Script: ared/wbooker/sentFlow-22.sh
☑ Is Flexible

Current Booking
lbi-03 from 2011-10-31 07:00:00 to 2011-10-31 09:00:00 CPU
lbi-05 from 2011-10-31 06:00:00 to 2011-10-31 08:00:00 CPU
lbi-11 from 2011-10-31 06:00:00 to 2011-10-31 08:00:00 CPU

Notes
A set of java apps, Hadoop jobs, and Vertica queries for running the sentiment analysis flow #08

Booking Id	User	Notes
854	lbiSA	A Hadoop/Vertica jobs for sentiment analysis flow
856	SRGadmin	Hadoop job: sort2-12
857	SentimentAnalysisUSer7	A java app

Figure 5: Manual reservation

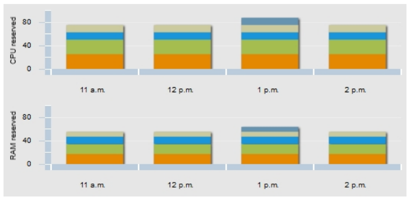
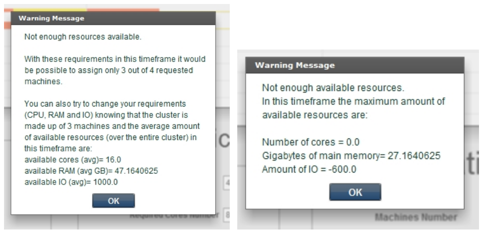


Figure 6: Booking information



Warning Message

Not enough resources available.

With these requirements in this timeframe it would be possible to assign only 3 out of 4 requested machines.

You can also try to change your requirements (CPU, RAM and IO) knowing that the cluster is made up of 3 machines and the average amount of available resources (over the entire cluster) in this timeframe are:
available cores (avg)= 16.0
available RAM (avg GB)= 47.1640625
available IO (avg)= 1000.0

OK

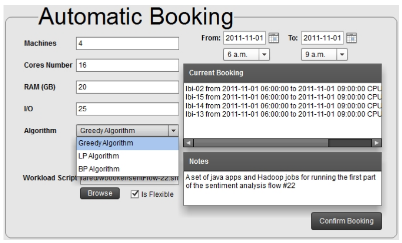
Warning Message

Not enough available resources.
In this timeframe the maximum amount of available resources are:

Number of cores = 0.0
Gigabytes of main memory= 27.1640625
Amount of IO = -600.0

OK

Figure 7: Warning messages



Automatic Booking

Machines: 4
Cores Number: 16
RAM (GB): 20
I/O: 25
Algorithm: Greedy Algorithm
Greedy Algorithm
LP Algorithm
BP Algorithm
From: 2011-11-01 6 a.m. To: 2011-11-01 9 a.m.
Workload Script: lared/wbooker/sentFlow-22.sh
☑ Is Flexible

Current Booking
lbi-02 from 2011-11-01 06:00:00 to 2011-11-01 09:00:00 CPU
lbi-15 from 2011-11-01 06:00:00 to 2011-11-01 09:00:00 CPU
lbi-14 from 2011-11-01 06:00:00 to 2011-11-01 09:00:00 CPU
lbi-13 from 2011-11-01 06:00:00 to 2011-11-01 09:00:00 CPU

Notes
A set of java apps and Hadoop jobs for running the first part of the sentiment analysis flow #22

Browse
Confirm Booking

Figure 8: Automatic reservation

monitoring functionality is offered by other tools like Ganglia[3]. However, our purpose is to provide an easy-to-install and easy-to-use tool to assist the reservation process. Users of *CloudAlloc* have confirmed that it is indeed a very handy choice for lightweight cluster monitoring.

Scenario 2: *CloudAlloc* offers two ways to submit a workload request, manual or automatic. Figure 5 shows our console for manual reservations. For a manual request, the user requests resources at specific nodes. To assist in node selection, the user may choose either a daily or weekly view and observe the system utilization per node. She can pick a time slot for a set of machines and add the requirements. For each time slot, one may see how a specific node has been reserved, i.e., booking information and graphs for system utilization. Figure 6 shows example bookings for a four hour period. Manual requests are checked for satisfiability. If the request can be satisfied, a corresponding entry for the booking is stored in the repository. The user may also choose to be notified by email for the booking. If the request cannot be satisfied, a detailed report explains to the user what requirement cannot be satisfied and also, she gets a list of available options that are close to her needs (see Figure 7). Alternatively, the user may choose to use the automatic reservation system. Then, she needs to specify the desired system requirements and a time window. Figure 8 shows the input form for automatic reservations (the console for automatic reservations is similar to Figure 5).

Scenario 3: For our third scenario, we show how the resource allocation algorithm may be changed (see Figure 8). In our demo, the greedy algorithm is the default, but *CloudAlloc* provides the option to select the optimal allocation algorithm. Then, the future workload reservations are reallocated and the differences, if any, may be observed.

Scenario 4: We show how a script can be associated with a workload to initiate execution of the workload. In this case, the workload script is invoked when the workload is scheduled for execution. A user may be notified by email when its workload starts/finishes its execution.

Scenario 5: For the fifth scenario we add/delete (virtual) nodes to/from our cluster and then, see how the monitoring system effortlessly updates the dashboard. We also reoptimize existing workloads under the new configuration showing how the system adapts to cluster changes.

In all scenarios, the participants will have the opportunity for a lively interaction. At the same time, we solicit feedback and comments for the continuation of our work.

[3]http://ganglia.sourceforge.net/

5. CONCLUSIONS

CloudAlloc has been implemented at HP Labs. During its operation, we have gathered several fruitful comments and feedback from our colleagues and due to them our system has been improved a lot. A more complete description of *CloudAlloc* is in [5], which is available upon request.

6. REFERENCES

[1] *Nimrod/G. Url: messagelab.monash.edu.au/NimrodG.*
[2] Platform LSF. http://www.platform.com/workload-management/high-performance-computing, 2011.
[3] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Y. Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *CoRR*, abs/1007.0066, 2010.
[4] C. Curino et al. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.
[5] E. Iori, A. Simitsis, T. Palpanas, S. Harizopoulos, and K. Wilkinson. CloudAlloc: Objective-based resource sharing in compute clusters. Technical Report, 2012.
[6] M. Jette and M. Grondona. SLURM: Simple linux utility for resource management. In *ClusterWorld*, 2003.
[7] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.
[8] L. Wang et al. Scientific cloud computing: Early definition and experience. In *IEEE HPCC*, pages 825–830, 2008.
[9] W. Zeng, Y. Zhao, and J. Zeng. Cloud service and service selection algorithm research. In *GEC*, pages 1045–1048, 2009.