Proceedings of the Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit

in conjunction with the
International Conference on Autonomic Computing 2012

San Jose, 21 September 2012

Conference Co-Chairs:

Michael Kozuch,
Gregor von Laszewski,
Robert Grossman,
Dejan Milojicic,
Rick McGeer

Sponsors:

Open Cirrus Consortium, the Open Cloud Consortium, and FutureGrid
Web Site:

http://fedcloud.cyberaide.org/

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York New York 10121-0701

ACM ISBN: 978-1-4503-1754-2

Message from the Program Co-Chairs

Welcome to the Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit held in conjunction with the International Conference on Autonomic Computing 2012 in San Jose, on 21 September 2012.

This workshop brings together researchers and practitioners to discuss the newest ideas and challenges in cloud services and federated cloud computing. The workshop consists of the presentation of peer reviewed papers to the workshop participants and the active discussion of topics related to the topic. Furthermore, we have planned a panel discussion.

The services offered by clouds are becoming critical for a wide variety of applications used by industry, education and government. There are now many examples of successful cloud services offered by public, private and community clouds. Many efforts exist that are creating cloud toolkits and frameworks to simplify the development and delivery of cloud services. The main purpose of this workshop is to bring together those responsible for designing, managing, and operating clouds services so that they can share experiences with each other. The workshop also welcomes users with requirements for new cloud services.

We are particularly interested in cloud services that can be used for federating clouds. Topics of interest include: Experiences, best practices, and lessons learned from operating cloud services; Testbeds for designing new cloud services; Cloud services for federating clouds; Management and provisioning of cloud services; Health and status monitoring of cloud services; Security of cloud services; Requirements for new cloud services; Reliability and fault tolerance of cloud services; Cloud services that span public and private clouds including the design of cloud services, Intercloud services, Federation services Identity services Cloud bursting services, Cloud services for emerging applications; Applications utilizing such services; Cloud Software and Tools for IaaS, PaaS, Hadoop, and others.

This workshop builds upon the success of the prior Open Cirrus events and the prior Open Cloud Consortium and FutureGrid events but was expanded to include other organizations. The goal is to help building a community for those responsible for operating clouds and cloud testbeds, as well as those interested in designing new cloud services.

The workshop could have not been conducted without the committee that provided the peer review for all the papers. We like to thank them for their help and list them here in alphabetical order: Brandic, Ivona; Desai, Narayan; Desprez, Frédéric; Diaz, Javier; Fitzgerald, Steve; Fox, Geoffrey; Gavrilovska, Ada; Grossman, Robert; Keahey, Kate; Kozuch, Michael; Llorente, Ignacio M.; Lucas-Simarro, Jose Luis; McGeer, Rick; Milojicic, Dejan; Ramakrishnan, Lavanya; Riedel, Morris; Toews, Everett; von Laszewski, Gregor. The conference and submission site was managed by Gregor von Laszewski.

Enjoy the conference!

Gregor von Laszewski,
Robert Grossman,
Dejan Milojicic,
Michael Kozuch,
Rick McGeer

Table of Contents

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit

21 September 2012, San Jose

Optimizing VM Placement for HPC in the Cloud

Abhishek Gupta*
Dejan Milojicic
HP Labs Palo Alto, CA, USA
(abhishek.gupta2, dejan.milojicic)@hp.com

Laxmikant V. Kalé
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kale@illinois.edu

ABSTRACT

"Computing as a service" model in cloud has encouraged High Performance Computing to reach out to wider scientific and industrial community. Many small and medium scale HPC users are exploring Infrastructure cloud as a possible platform to run their applications. However, there are gaps between the characteristic traits of an HPC application and existing cloud scheduling algorithms. In this paper, we propose an HPC-aware scheduler and implement it atop Open Stack scheduler. In particular, we introduce topology awareness and consideration for homogeneity while allocating VMs. We demonstrate the benefits of these techniques by evaluating them on a cloud setup on Open Cirrus testbed.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; K.6.4 [**System Management**]: Centralization/decentralization

Keywords

High Performance Computing, Clouds, Resource Scheduling

1. INTRODUCTION

Cloud computing is increasingly being explored as a cost effective alternative and addition to supercomputers for some HPC applications [8, 13, 17, 24]. Cloud provides the benefits of economy of scale, elasticity and virtualization to HPC community and is attracting many users which cannot afford to establish their own dedicated cluster due to up-front investment, sporadic demands or both.

However, presence of commodity interconnect, performance overhead introduced by virtualization and performance variability are some factors which imply that cloud can be suitable for some HPC applications but not all [8, 15]. Past research [15,16,18,24] on HPC in cloud has primarily focused on evaluation of scientific parallel applications (such as those written in MPI [6]) and have been mostly pessimistic. To the best of our knowledge, there have been few efforts on VM scheduling algorithms which take into account the nature of HPC application - tightly coupled processes which perform frequent inter-process communication and synchronizations. VM to physical machine placement can have a significant impact on performance. With this as motivation, the primary question that we address through this research is the following: Can we improve HPC application performance in Cloud through intelligent VM placement strategies tailored to HPC application characteristics?

Current cloud management systems such as Open Stack [3] and Eucalyptus [23] lack an intelligent scheduler for HPC applications. In our terminology, *scheduling or placement* refers to selection of physical servers for provisioning virtual machines. In this paper, we explore the challenges and alternatives for scheduling HPC applications on cloud. We implement HPC-aware VM placement strategies - specifically topology awareness and hardware awareness in Open Stack scheduler and evaluate their effectiveness using performance measurement on a cloud, which we setup on Open Cirrus [9] platform.

The benefit of our approach is that HPC-aware scheduling strategies can result in significant benefits for both HPC users and cloud providers. Using these strategies, cloud providers can utilize infrastructure more and offer improved performance to cloud users. This can allow cloud providers to obtain higher profits for their resources. They can also pass some benefits to cloud users to attract more customers.

The key contribution of this work is a novel scheduler for HPC application in cloud for Open Stack through topology and hardware awareness.We address the initial VM placement problem in this paper.

The remainder of this paper is organized as follows: Section 2 provides background on Open Stack. Section 3 discusses our algorithms followed by implementation. Next, we describe our evaluation methodology in Section 4. We present performance results in Section 5. Related work is discussed in section 6. We give concluding remarks along with an outline of future work in Section 7.

2. OPEN STACK SCHEDULER

Open Stack [3] is an open source cloud management system which allows easy control of large pools of infrastructure resources (compute, storage and networking) through-

*Abhishek, an intern at HP labs, is also a Ph.D. student at University of Illinois at Urbana-Champaign.

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, September 21, 2012, San Jose, CA, USA.

out a datacenter. Open Stack has multiple projects, each with a different focus, examples are compute (Nova), storage (Swift), Image delivery and registration (Glance), Identity (Keystone), Dashboard (Horizon) and Network Connectivity (Quantum).

Our focus in this work is on the compute component of Open Stack, known as Nova. A core component of a cloud setup using Open Stack is the Open Stack scheduler, which selects the physical nodes where a VM will be provisioned. We implemented our scheduling techniques on top of existing Open Stack scheduler and hence first we summarize the existing scheduler. In this work, we used the Diablo (2011.3) version of Open Stack.

Open Stack scheduler receives a VM provisioning request (`request_spec`) as part of RPC message. `request_spec` specifies the number of instances (VMs), instance type which maps to resource requirements (number of virtual cores, amount of memory, amount of disk space) for each instance and some other user specified options that can be passes to the scheduler at run time. Host capability data is another important input to the scheduler which contains the list of physical servers with their current capabilities (free CPUs, free memory etc.).

Using `request_spec` and capabilities data, the scheduling algorithm consists of two steps:

1. Filtering - excludes hosts which are incapable of fulfilling the request based on certain criteria (e.g free cores < requested virtual cores).
2. Weighing - computes the relative fitness of filtered list of hosts to fulfill the request using cost functions. Multiple cost functions can be used, each host is first scored by running each cost function and then weighted scores are calculated for each host by multiplying score and weight of each cost function. An example of cost function is free memory in a host.

Next, the list of hosts is sorted by the weighted score and VMs are provisioned on hosts using this sorted list.

There are various filtering and weighing strategies currently available in Open Stack. However, one key disadvantage of the current Open Stack scheduler is that scheduling policies do not consider application type and priorities, which could allow more intelligent decision making. Further, scheduling policies ignore processor heterogeneity and network topology while selecting hosts for VMs. Existing scheduling policies consider the k VMs requested as part of a user request as k separate VM placement problems. Hence, it runs the core of scheduling algorithm k times to find placement of k VMs constituting a single request, thereby avoiding any co-relation between the placement of VMs which comprise a single request. Some example of currently available schedulers are *Chance* scheduler (chooses host randomly across availability zones), *Availability zone* scheduler (similar to chance, but chooses host randomly from within a specified availability zone) and *Simple* scheduler (chooses least loaded host e.g. host with least number of running cores).

3. AN HPC-AWARE SCHEDULER

In this paper, we address the initial VM placement problem (Figure 1). The problem can be formulated as - Map: k VMs $(v_1, v_2, .., v_k)$ each with same, fixed resource requirements (decided by instance type: CPU, memory, disk etc)

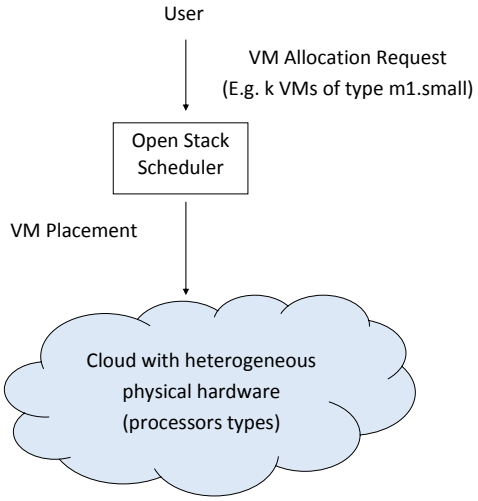


Figure 1: VM Placement

to N physical servers $P_1, P_2, ..., P_n$, which are unoccupied or partially occupied, while satisfying resource requirements. Moreover, our focus is on providing the user a VM placement optimized for HPC.

Next, we discuss the design and implementation of the proposed techniques atop existing Open Stack scheduling framework.

3.1 Techniques

In this section, we describe two techniques for optimizing the placement of VMs for an HPC-optimized allocation.

3.1.1 Topology Awareness

An HPC application consists of n parallel processes, which typically perform inter-process communication for overall progress. The effect of cluster topology on application performance has been widely explored by HPC community. In the context of cloud, the cluster topology is unknown to the user. The goal is to place the VMs on those physical machines which are as close to each other as possible with respect to the cluster topology. Let us consider a practical example - the cluster topology of Open Cirrus HP Labs site - a simple topology, each server is a 4-core node and there are 32 nodes in a rack, all nodes in a rack are connected by a 1Gbps link to a switch. All racks are connected using a 10Gbps link to a top-level switch. In this case, the 10Gbps link is shared by 32 nodes, effectively providing a bandwidth of 10Gbps/32 = 0.312 Gbps between two nodes in different rack when all nodes are communicating. However, the point-to-point bandwidth between two nodes in the same rack is 1 Gbps. Hence, it would be better to pack VMs to nodes in the same rack compared to a random placement policy, which can potentially distribute them all over the cluster.

3.1.2 Hardware Awareness/Homogeneity

Another characteristics of HPC applications is that they are generally iterative and bulk synchronous, which means that in each iteration, there are two phases - computation followed by communication/synchronization which also acts as a barrier. The next iteration can start only when all processes have finished previous iteration. Hence, a single slow process can degrade the performance of whole application. This also means that faster processors waste lot of time wait-

ing for slower processors to reach the synchronization point.

In case of cloud, the user is unaware of the underlying hardware on which his VMs are placed. Since clouds evolve over time, they consist of heterogeneous physical servers. Existing VM placement strategies ignore the heterogeneity of the underlying hardware. This can result in some VMs running on faster processors, while some running on slower processors. Some cloud providers, such as Amazon EC2 [1], address the problem of heterogeneity by creating a new compute unit and allocating based on that. Using a new compute unit enables them to utilize the remaining capacity and allocating it to a separate VM using shares (e.g 80-20 CPU share). However, for HPC applications, this can actually make the performance worse, since all the processes comprising an application can quickly become out of sync due to the effect of other VM on same node. To make sure the k VMs are at sync, all k VMS need to be scheduled together if they are running on heterogeneous platform and sharing CPU with other VMs (some form of gang scheduling). In addition, the interference arising from other VMs can have a significant performance impact on HPC application. To avoid such interference, Amazon EC2 uses a dedicated cluster for HPC [2]. However, the disadvantage of this is lower utilization which results in higher price.

To address the needs of homogeneous hardware for HPC VMs, we take an alternative approach. We make the VM placement hardware aware and ensure that all k VMs of a user request are allocated same type of processors.

3.2 Implementation

We implemented the techniques discussed in section 3.1 on top of Open Stack scheduler. The first modification to enable HPC-aware scheduling is to switch to the use of group scheduling which allows the scheduling algorithm to consider placement of k VMs as a single scheduling problem rather than k separate scheduling problems. Our topology-aware algorithm (pseudo-code shown in Algorithm 1) proceeds by considering the list of physical servers (*hosts* in Open Stack scheduler terminology). Next, the filtering phase of the scheduler removes the hosts which cannot meet the requested demands of a single VM. We also calculate the maximum number of VMs each host can fit (based on the number of virtual cores and amount of memory requested by each VM). We call it `hostCapacity`. Next, for each rack, we calculate the number of VMs the rack can fit (`rackCapacity`) by summing the `hostCapacity` for all hosts in a rack. Using this information, scheduler creates a build plan, which is an ordered list of hosts such that if $i < j$, i^{th} host belongs to a rack with higher capacity compared to j^{th} host or both host belong to same rack and `hostcapacity` of i^{th} host is greater or equal to that of j^{th} host. Hence, the scheduler places VMs in a rack with largest `rackCapacity` and the host in that rack with largest `hostCapacity`.

One potential disadvantage of our current policy of selecting the rack with maximum available capacity is unnecessary system fragmentation. To overcome this problem, we plan to explore more intelligent heuristics such as selecting the rack (or a combination of racks) with the minimum excess capacity over the VM set allocation.

To ensure homogeneity, the scheduler first groups the hosts into different lists based on their processor type and then applies the scheduling algorithm described above to these groups, with preference to the best configuration first. Currently, we use CPU frequency as the distinction criteria between different processor types. For more accurate distinction, we plan on incorporating additional factors such as cache sizes and MIPS.

Algorithm 1 Pseudo code for Topology aware scheduler

```
capability = list of capabilities of unique hosts
request_spec = request specification
numHosts = capability.length()
filteredHostList = new vector < int >
rackList = new set < int >
hostCapacity = new int[numHosts]
for i = 1 to i < numHosts do
    hostCapacity[i] = max
    (capability[i].freeCores/request_spec.instanceVcpus,
    capability[i].freeMemory/request_spec.instanceMemory)
    if hostCapacity[i] > 0 then
        filteredHostList.push(i)
    end if
    rackList.add(capability[i].rackid)
end for
numRacks = rackList.length()
rackCapacity = new int[numRacks]
for j = 1 to j < numRacks do
    rackCapacity[j] = sum_i hostCapacity[i]  ∀i such that
    capability[i].rackid = j
end for
Sort filteredHostList by decreasing order of hostCapacity[j]
where j ∈ filteredHostList. Call this sortedHostList
Stable Sort sortedHostList by decreasing order of rackCapacity[capability[j].rackid] where
j ∈ filteredHostList. Call this PrelimBuildPlan.
buildPlan = new vector[int]
for i = 1 to i <= numFilteredHosts do
    for j = 1 to j <= hostCapacity[PrelimBuildPlan[i]] do
        buildPlan.push(PrelimBuildPlan[i])
    end for
end for
return buildPlan
```

4. EVALUATION METHODOLOGY

In this section, we describe the platform which we setup and the applications which we chose for this study.

4.1 Experimental Testbed

We setup a cloud using Open Stack on Open Cirrus testbed at HP Labs site [9]. Open Cirrus is a cluster established for system level research. This testbed has 3 types of servers:

- Intel Xeon E5450 (12M Cache, 3.00 GHz)
- Intel Xeon X3370 (12M Cache, 3.00 GHz)
- Intel Xeon X3210 (8M Cache, 2.13 GHz)

The topology is as described in section 3.1.1.

We used KVM [7] as hypervisor since past research has indicated that KVM is a good choice for virtualization for HPC clouds [25]. For network virtualization, we initially used the default network driver (*rtl8139*) but subsequently switched to the *virtio-net* driver on observing improved network performance (details in section 5). The VMs used for the experiments performed in this paper were of the type m1.small (1 core, 2 GB memory, 20 GB disk).

4.2 Benchmarks and Applications

For this study, we chose certain benchmarks and real world applications as representatives of HPC applications.

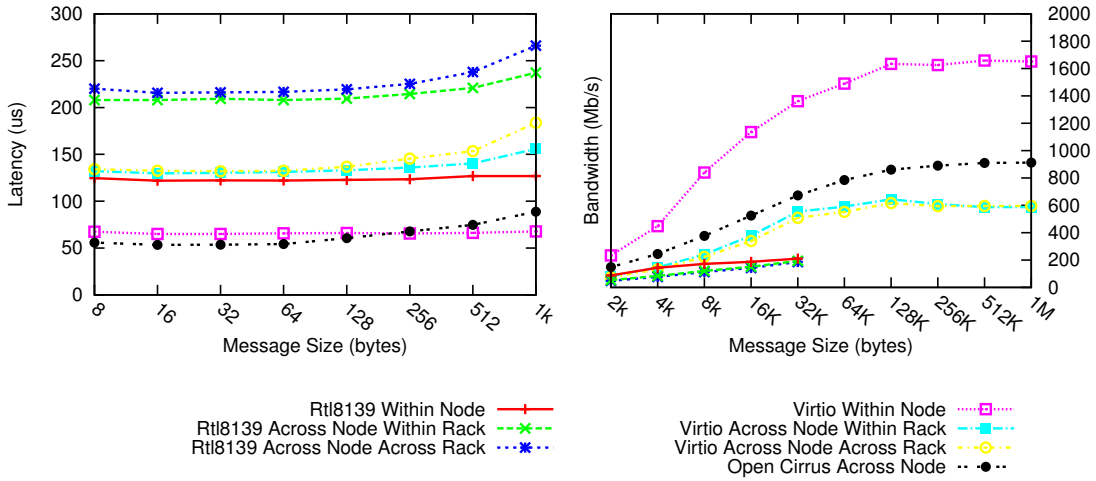


Figure 2: Latency and Bandwidth vs. Message Size for different VM placement.

- Jacobi2D - A kernel which performs 5-point stencil computation to average values in a 2-D grid. Such stencil computation is very commonly used in scientific simulations, numerical algebra, and image processing.
- NAMD [11] - A highly scalable molecular dynamics application and representative of a complex real world application used ubiquitously on supercomputers. We used the ApoA1 input (92k atoms) for our experiments.
- ChaNGa [19] (Charm N-body GrAvity solver) - A scalable application used to perform collisionless N-body simulation. It can be used to perform cosmological simulation and also includes hydrodynamics. It uses Barnes-Hut tree [10] to calculate forces between particles. We used a 300, 000 particle system for our runs.

All these applications are written in Charm++ [20], which is a C++ based object-oriented parallel programming language. We used the `net-linux-x86-64 udp` machine layer of Charm++ and used -O3 optimization level.

5. RESULTS

We first evaluate the effect of topology-aware scheduling. Figure 2 shows the results of a ping-pong benchmark. We used a Converse [12] (underlying substrate of Charm++) ping-pong benchmark to compare latencies and bandwidth for various VM placement configurations. Figure 2 presents several insights. First, we see that *virtio* outperforms *rtl8139* network driver both for intra-node and inter-node VM communication, making it a natural choice for remainder of the experiments. Second, there is significant virtualization overhead. Even for communication between VMs on same node, there is a 64 usec latency using virtio. Similarly, for inter-node communication, VM latencies are around twice compared to communication between physical nodes in Open Cirrus and there is also substantial reduction in achieved bandwidth, although the degradation in bandwidth (33% reduction) is less compared to the degradation in latencies (100% increase). Third, there is very little difference for latencies and bandwidth when comparing communication between VMs on different nodes but same rack and between VMs on different nodes on different racks. This can be attributed to the use of wormhole routing in modern network

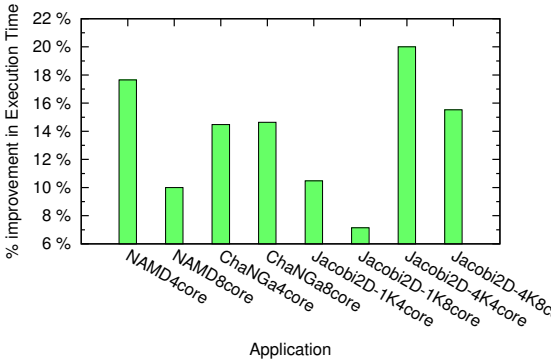


Figure 3: Percentage improvement achieved using hardware aware Scheduling compared to the case where 2 VMs were on slower processors and rest on faster processors

which means that the extra hops cause very little performance overhead. As we discussed in section 3.1.1, effects of intra-rack and cross-rack communication become more prominent as we scale up or the application performs significant collective communication such as all-to-all data movement.

We compared our topology aware scheduler with random scheduling. To perform a fair comparison, we explicitly controlled the scheduling such that the default (random) case corresponds to a mapping where the VMs are distributed to two racks, we keep the number of VM on each host as two in both cases. For the topology aware case, the scheduler placed all VMs to the hosts in same rack. For these experiments, we were able to gain up to 5% in performance compared to random scheduling. We expect the benefits of topology aware scheduling to increase as we run on higher core counts in cloud.

Figure 3 shows the effect of hardware aware VM placement on the performance of some applications for different number of VMs. We compare two cases - when all VMs are mapped to same type of processors (Homo) and when two VMs are mapped to a slower processors, rest to the faster processor (Hetero). To perform a fair comparison, we keep the number of VMs on each host as two in both cases. % im-

provement is calculated as $(T_{Hetero} - T_{Homo})/T_{Hetero}$. From Figure 3, we can observe that the improvement achieved depends on the nature of application and the scale at which it is run. The improvement is not equal to the ratio of sequential execution time on slower processor to that on faster processor since parallel execution time also includes the communication time and parallel overhead, which is not necessarily dependent on the processor speeds. We achieved up to 20% improvement in parallel execution time - which means we were able to save *20% of time * N CPU-hours*, where N is the number of processors used.

We used the Projections [21] tool to analyze the performance bottlenecks in these two cases. Figure 4 shows the CPU (VM) timelines for an 8-core Jacobi2D experiment, x-axis being time and y-axis being the (virtual) core number. White portion shows idle time while colored portions represent application functions. For figure 4a, the first 2 VMs were mapped to slower processors and are busy for all the time. It is clear that there is lot more idle time on VMs 3-7 compared to first 2 VMs since they have to wait for VMs 0-1 to reach the synchronization point after finishing the computation. The first two VMs are bottleneck in this case and result in execution time being 20% more compared to the homogeneous case. In Figure 4b all 8 VMs are on same type of processors, here the idle time is due to the communication time.

Figure 5 shows the overall performance that we achieve using these techniques on our test-bed. We compare the performance to that achieved without virtualization on the same testbed. It is clear that using our techniques, even communication intensive applications such as NAMD and ChaNGa scale quite well, compared to their scalability on the physical platform. However, effect of virtualization on network performance can quickly become a scalability bottleneck (as suggested by Figures 2 and 4).

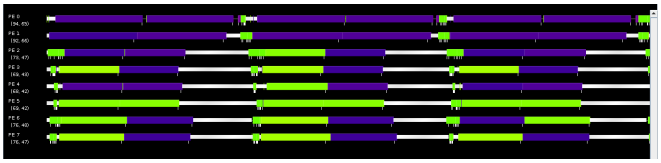
6. RELATED WORK

There have been several studies on HPC in cloud using benchmarks such as NPB and real applications [8, 13, 15–17, 22, 24]. The conclusions of these studies have been the following:

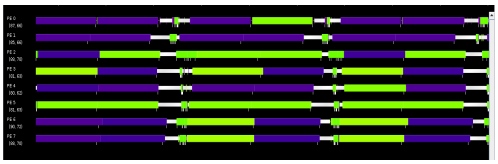
- Interconnect and I/O performance on commercial cloud severely limit performance and cause significant performance variability.
- Cloud cannot compete with supercomputers based on the metric $/GFLOPS for large scale HPC applications.
- It can be cost-effective to run some applications on cloud compared to supercomputer, specifically those with less communication and at low scale.

In our earlier work [15], we studied the performance-cost tradeoffs of running different applications on supercomputer vs. cloud. We demonstrated that the suitability of a platform for an HPC application depends upon application characteristics, performance requirements and user preferences. In another work, we explored techniques to improve HPC application performance in Cloud through an optimized parallel run-time system. We used a cloud-friendly load balancing system to reduce the performance degradation suffered by parallel application due to effect of virtualization in cloud.

In this paper, we take one step further and research VM placement strategies which can result in improved application performance. Our focus is HPC applications - which consist of k parallel instances typically requiring synchronization through inter process communication.



(a) Heterogeneous: First two VMs on slower processors



(b) Homogeneous: All 8 VMs on same type of processors

Figure 4: Timeline of 8 VMs running Jacobi2D (2K by 2K): white portion shows idle time while colored portions represent application functions.

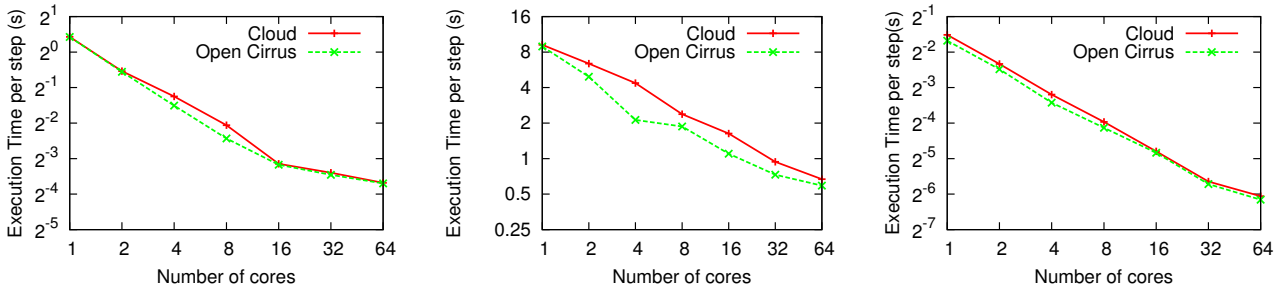
Existing scheduler do not address this problem. Cloud management systems such as Open Stack [3], Eucalyptus [23] and Open Nebula [5] lack strategies which consider such tightly coupled nature of VMs comprising a single user request, while making scheduling decisions. Fan et al. discuss topology aware deployment for scientific applications in cloud and map the communication topology of a parallel application to the VM physical topology [14]. However, we focus on allocating VMs in a topology aware manner to provide a good set of VMs to an HPC application user. Moreover, we address the heterogeneity of hardware. Amazon EC2's Cluster Compute instance introduces Placement Groups such that all instances launched within a Placement Group are expected to have low latency and full bisection 10 Gbps bandwidth between instances [2]. It is unknown and undisclosed how strict those guarantees are and what techniques are used to provide them.

There have been several efforts on job scheduling for HPC applications, such as LSF (Load Sharing Facility) [4] - a commercial job scheduler which allows load sharing using distribution of jobs to available CPUs in heterogeneous network. SLURM, ALPS, MOAB, Torque, Open PBS, PBS Pro, SGE, Condor are other examples. However, they perform scheduling at the granularity of physical machines. In Cloud, virtualization enables consolidation and sharing of nodes between different types of VMs which can enable improved server utilization.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we learned that utilizing the knowledge of target application for a VM can lead to more intelligent VM placement decisions. We made a case for HPC-aware VM placement techniques and demonstrated the benefits of using HPC-aware VM placement techniques for efficient execution of HPC applications in cloud. In particular, we implemented topology and hardware awareness in Open Stack scheduler and evaluated them on a cloud setup on Open Cirrus testbed.

In future, we plan to research how to schedule a mix of HPC and non-HPC applications in an intelligent fashion to increase resource utilization. E.g. co-locating VMs



(a) NAMD (Molecular Dynamics) (b) ChaNGa (Cosmology) (c) Jacobi2D - 4K by 4K matrix

Figure 5: Execution Time vs. Number of cores/VMs for different applications.

which are network bandwidth intensive and VMs which are compute intensive to increase resource utilization. Comparison with other network virtualization techniques such as vhost_net and tcp protocol is another direction of future research.

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`.

[2] High Performance Computing (HPC) on AWS. `http://aws.amazon.com/hpc-applications`.

[3] Open Stack Open Source Cloud Computing Software. `http://openstack.org`.

[4] Platform Computing. `http://www.platform.com/workload-management/high-performance-computing`.

[5] The Cloud Data Center Management Solution . `http://opennebula.org`.

[6] Mpi: A message passing interface standard. In *M. P. I. Forum*, 1994.

[7] KVM – Kernel-based Virtual Machine. Technical report, Redhat, Inc., 2009.

[8] Magellan Final Report. Technical report, U.S. Department of Energy (DOE), 2011. `http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf`.

[9] A. Avetisyan et al. Open Cirrus: A Global Cloud Computing Testbed. *IEEE Computer*, 43:35–43, April 2010.

[10] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[11] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *IPDPS 2008*, pages 1–12, April 2008.

[12] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.

[13] C. Evangelinos and C. N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. Cloud Computing and Its Applications, Oct. 2008.

[14] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu. Topology-aware deployment of scientific applications in cloud computing. *Cloud Computing, IEEE International Conference on*, 0, 2012.

[15] A. Gupta and D. Milojicic. Evaluation of HPC Applications on Cloud. In *Open Cirrus Summit (Best Student Paper)*, pages 22 –26, Atlanta, GA, Oct. 2011.

[16] A. Gupta et al. Exploring the performance and mapping of hpc applications to platforms in the cloud. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 121–122, New York, NY, USA, 2012. ACM.

[17] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Trans. Parallel Distrib. Syst.*, 22:931–945, June 2011.

[18] K. Jackson et al. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *CloudCom'10*, 2010.

[19] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *IPDPS 2008*, pages 1–12, 2008.

[20] L. Kalé. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.

[21] L. Kalé and A. Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, Apr. 1993.

[22] J. Napper and P. Bientinesi. Can Cloud Computing Reach the Top500? In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, UCHPC-MAW '09, pages 17–20, New York, NY, USA, 2009. ACM.

[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Proceedings of Cloud Computing and Its Applications*, Oct. 2008.

[24] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *LOGIN*, pages 18–23, 2008.

[25] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox. Analysis of virtualization technologies for high performance computing environments. *Cloud Computing, IEEE International Conference on*, 0:9–16, 2011.

Virtualizing HPC applications using modern hypervisors

Alexander Kudryavtsev, Vladimir Koshelev, Boris Pavlovic and Arutyun Avetisyan
Institute for System Programming, Russian Academy of Sciences
109004, Moscow, Alexander Solzhenitsyn st., 25
Russian Federation
{alexk,vedun,bpavlovich,arut}@ispras.ru *

ABSTRACT

In this paper we explore the prospects of virtualization technologies being applied to high performance computing tasks. We use an extensive set of HPC benchmarks to evaluate virtualization overhead, including HPC Challenge, NAS Parallel Benchmarks and SPEC MPI2007. We assess KVM and Palacios hypervisors and, with proper tuning of hypervisor, we reduce the performance degradation from 10-60% to 1-5% in many cases with processor cores count up to 240. At the same time, a few tests provide overhead ranging from 20% to 45% even with our enhancements.

We describe the techniques necessary to achieve sufficient performance. These include host OS tuning to decrease noise level, using nested paging with large pages for efficient guest memory allocation, and proper NUMA architecture emulation when running virtual machines on NUMA hosts.

Comparing KVM/QEMU and Palacios hypervisors, we conclude that in general the results with proper tuning are similar, with KVM providing more stable and predictable results while Palacios being much better on fine-grained tests at a large scale, but showing abnormal performance degradation on a few tests.

Categories and Subject Descriptors: D.4.7 [Operating Systems]: Organization and Design

General Terms: Design, Experimentation, Measurement, Performance.

Keywords: Virtual machine monitors, high performance computing, parallel computing.

1. INTRODUCTION

Virtualization technologies are getting more mature and their capabilities have grown rapidly in the last few years. Virtualization gets new applications in different areas. Hardware-assisted virtualization technologies are getting capable of providing near-native performance for some classes of applications. As a result, researchers started to investigate the capabilities and limitations of virtualization when applied to High Performance Computing (HPC) tasks.

Benefits of applying virtualization into HPC area are being widely discussed [17, 7]. Fault tolerance, compatibility and flexibility are among them. Another idea is to employ cloud concept for building HPC clouds which provide scalability, cost effectiveness and ease of access. Recent research shows that HPC virtualization is feasible for at least some classes of applications [17].

Currently there is a substantial lack of experimental data in HPC virtualization area. Different applications have to be tested on different hardware using a variety of hypervisors to fully understand the limitations of existing virtualization technologies. Modern multi-socket hardware provide additional requirements to virtualization software, including Non-Uniform Memory Access (NUMA) emulation in guest system. Proper NUMA emulation is really important for VM performance when running on multi-socket NUMA hardware because of different memory access latencies for CPU accessing its local memory and other CPUs' memory.

In this work we explore the performance of HPC applications running inside a set of VMs in a cluster. Our primary goal is to evaluate full-system virtualization overhead, to understand the reasons of this overhead, and to minimize it if possible. We assess Kernel Virtual Machine (KVM) [9] and Palacios hypervisor [11], which was developed specially for HPC systems. It should be noted that we used a modified version of Palacios since the original version does not yet supports many features required to launch it on our test hardware. The changes we made to Palacios are briefly described in the following sections.

To achieve maximum performance compared to the native case, we allocate virtual machines (VMs) as much resources as possible, including all processor cores, and pass high-speed interconnect device into VM using Intel VT-d in KVM and paravirtualization in Palacios. Also we expose an emulated NUMA architecture into the VM and set it up to reflect the real NUMA configuration of our test hardware.

We use HPC Challenge (HPCC) benchmark [12], NAS Parallel Benchmarks (NPB) [4] and SPEC MPI2007 [13] benchmark as a test suite since these benchmarks are widely used as HPC system performance indicators. Performance tests were made on HPC clusters containing 8 and 20 Intel Xeon nodes (up to 240 processor cores used) connected to a 40 Gb/s Infiniband network. Many previous results were gathered on single node systems, which does not fully evaluate virtualization overhead.

*The work was supported by The Ministry of education and science of Russia under the contract No. 07.524.11.4018.

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, September 21, 2012, San Jose, CA, USA.

Our main contributions are the following:

- We describe the techniques necessary to achieve sufficient performance using KVM/QEMU. These include host OS tuning to decrease noise level, using nested paging with large pages for efficient guest memory allocation, and proper NUMA architecture emulation when running virtual machines on NUMA hosts.
- We provide virtualization overhead test results for an extensive set of HPC applications.
- We compare the performance of Kitten HPC OS / Palacios hypervisor and Linux / KVM/QEMU hypervisor. We show that KVM/QEMU provides more stable and predictable results, while Palacios is much better on fine-grained tests, especially at large scale.
- We examine gathered test data and investigate the reasons of observed overhead caused by virtualization.

The remainder of this document is organized as follows. In the next section, the related work is discussed. Section 3 describes hypervisor systems which we use and some basic methods for performance optimization. Section 4 describes performance evaluation methodology and provides test results together with discussion. Section 5 contains conclusions.

2. RELATED WORK

A thorough analysis of present work in the area of HPC virtualization research is done in the paper by Andrew J. Younge et al. [17]. The authors note that current performance test results available in articles are sometimes conflicting with each other, and try to perform unbiased assessment of modern virtualization technologies, including Xen [5], KVM, VirtualBox [16]. The authors used HPCC and SPEC OpenMP benchmark suites, performance loss for High Performance Linpack (HPL) is about 30% on 8-core system. It should be noted that the authors performed all tests on a single node system, which does not allow one to evaluate performance loss while scaling number of nodes.

The authors of [17] point out that virtualization is feasible for at least some HPC applications and choose KVM as well-suited hypervisor for such tasks. From our experience, KVM is becoming one of the most full-featured open source hypervisor. When compared to Xen, KVM is much easier to manage and understand since it is built into mainline Linux kernel.

Regola and Ducom evaluate the I/O performance of KVM, Xen, OpenVZ [1] and Amazon's EC2 "Cluster Compute Node" [15]. The authors note that CPU virtualization performance is already studied well while I/O virtualization should be studied more thoroughly and in the paper the performance of disk and network I/O is evaluated. Also additional overhead of Intel's VT-d when passing Infiniband Host Channel Adapter (HCA) into VM is evaluated for the first time. NAS Parallel Benchmark MPI results show that the worst overhead for KVM is 30 times compared to the native case, for Xen — 50%. It seems that the authors used some outdated version of KVM/QEMU because our results are more optimistic.

OpenVZ I/O performance with Infiniband was not tested since it does not have Infiniband driver, but the authors claim that it has near-native I/O performance in almost all cases. We agree with that, since container-based virtualization is much more lightweight than full-system virtualization, but we believe that hardware-assisted virtualization technologies will evolve together with software to provide near-native I/O performance.

For example, as Lange et al. note [11], the main reason for performance overhead of device passed inside VM is the interrupt overhead. On every interrupt, CPU has to transfer control to the hypervisor. When the guest finishes interrupt handling, the end of interrupt signal is also handled by the hypervisor. This chain may significantly delay interrupt delivery, and it increases the latency. Gordon et al. proved this assumption in their Exit-Less Interrupt (ELI) system [8], which allows to pass interrupts inside VM without exit to the hypervisor. The authors' results show that ELI can decrease performance overhead of KVM/QEMU from 40 to 1-2% when running one core VM with Netperf, Apache and Memcached applications used as a benchmark. This performance overhead is caused by a huge number of interrupts issued by passed inside VM Ethernet card.

Virtualization of a large scale supercomputer was studied by Lange et al. [11] for the first time using a specially crafted Palacios hypervisor together with Kitten HPC OS [10] used as a host OS. Palacios hypervisor is developed as a part of the V3VEE project [3]. The authors gathered test results for a list of HPC applications and benchmarks, and the virtualization overhead was less than 5% with node count up to 4096 nodes. However it should be noted that these results were gathered with only one virtual CPU per one 4-core node.

The authors of Palacios also note the importance of OS noise problem. The impact of OS noise on parallel applications is being widely studied [6, 14]. In general, the noise impact is highly dependent on application computation-communication ratio, communication granularity, process grid characteristics, and for some applications even minor noise can lead to significant performance and scalability degradation. Thus, when virtualizing an HPC system, the host OS noise problem should be taken into account. From this point of view, Kitten OS should perform better as a host OS than Linux since it was designed specially for HPC.

3. VIRTUALIZATION SYSTEMS UNDER CONSIDERATION

The most widespread virtualization systems, in general, provide almost the same functionality. Xen and KVM seem to be the most popular hypervisors (among open source solutions) which are heavily used in production. As it was noted before, we decided to investigate KVM as a hypervisor for HPC applications. In the future work we hope to evaluate Xen too. Also we studied the Palacios hypervisor, which is created for computer architecture research and use in high performance computing.

KVM/QEMU

KVM hypervisor is a full virtualization solution for Linux which supports hardware-assisted virtualization extensions (Intel VT-x and AMD-V). KVM consists of several loadable kernel modules and provides only hardware-virtualized CPUs. The rest of the VM is implemented by QEMU emulator. QEMU can run VMs using KVM hardware-assisted

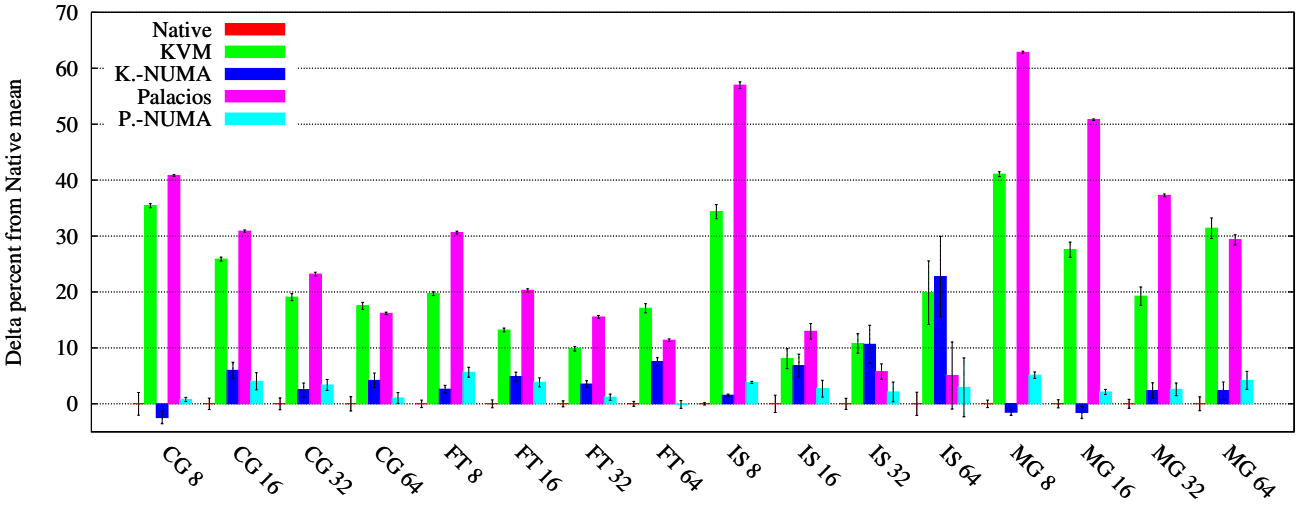


Figure 1: KVM/QEMU and Palacios results for NPB CG, FT, MG, IS tests.

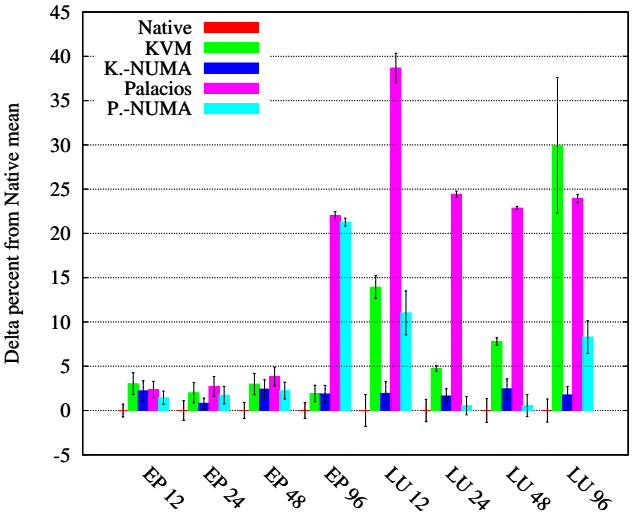


Figure 2: KVM/QEMU and Palacios results for NPB EP, LU tests.

virtualization or binary translation technique.

KVM/QEMU together with Linux allows real devices to be assigned to (or passed into) the VM. To support direct device assignment, the host hardware must contain IOMMU (I/O Memory Management Unit) — either Intel's VT-d or AMD's IOMMU. The main goal of the IOMMU is to map the device address space into guest physical address space using page tables. To allow guest system to use Infiniband HCA, we pass it into VM using Intel's VT-d.

The x86_64 virtual memory system has support for different page sizes. Linux kernel uses 4KB pages by default and contains feature called HugeTLB to provide larger pages (or *huge* pages) on demand. Huge pages may be used to decrease both the number of memory accesses required for guest physical address (PA) to host PA translation and the number of TLB misses when using nested paging.

The Linux kernel supports two ways of using HugeTLB: Transparent Hugepages and HugeTLBfs. The Transparent Hugepages mechanism allows the kernel to allocate anonymous memory using huge pages without the need to modify the application if the requested memory size is page size aligned. HugeTLBfs uses reserved huge pages via file system. Custom programs can map files from HugeTLBfs and these mappings will be backed by huge pages. The version of QEMU which we use (1.0.1) allocates memory for VM via the call to `posix_memalign` with 2MB alignment, thus Transparent Hugepages mechanism works by default.

Expose the real NUMA topology to VM.

A virtual SMP system running on top of a real NUMA system may suffer from serious performance degradation. Each NUMA node has its own CPU set and memory ranges and access from one node's CPU to other node's memory requires more time than access to the same node's memory. QEMU has support for NUMA system emulation, but emulated structure does not correspond to the real hardware topology. To solve this problem we pin virtual CPUs to distinct physical cores to disallow QEMU's threads migration between cores and employ `mbind` system call to be sure that selected memory ranges of VM memory will be allocated at corresponding nodes. We found that `mbind` ruins Transparent Hugepages allocation, so we allocate memory directly from HugeTLBfs using QEMU's `-mempath` parameter.

Palacios

The Palacios hypervisor [11] was developed with the goal to effectively virtualize HPC applications. The distinguishing feature of Palacios is its embeddable nature achieved via a set of unified host OS interfaces. Initially Palacios was embedded into Kitten HPC OS, but in the latest release Linux support is implemented too. We decided to assess Palacios since we believe it is a promising direction. We use Kitten OS as a host OS since this system should generate much less noise when compared to Linux – this advantage is the most interesting for our experiments.

Palacios supports nested paging and a number of shadow paging techniques. For our tests, we used nested paging with 2MB pages since it is the best choice for the Linux guest [11].

Device assignment technique.

The real communication device should be passed into VM to achieve performance which is comparable to the native

run. To make Palacios device assignment possible, special paravirtual interface is used. The memory for guest system is allocated in one physically contiguous range. The guest system can use hypercall to get its physical memory offset in the real physical memory. Using this offset, guest OS can patch addresses for DMA transactions of selected devices. In fact, changes required in guest OS to implement this interface are relatively simple. We created our own patch for the Linux kernel.

Since Kitten OS does not provide drivers for the most hardware, we had to pass some devices among Infiniband HCA inside the VM, namely SATA hard disk controller and Ethernet card.

Problems with Palacios.

As soon as we started our experiments with Palacios release 1.2, we faced some difficulties, including incomplete device assignment support, incomplete VT-x support, maximum $\sim$ 3.5GB of RAM, no NUMA support. We created our local branch and implemented these features which were required to launch VM on our test hardware[1].

One serious problem that was not solved at all is the problem with timing in the guest system due to the poor timer implementation in Palacios. Currently the guest clock is inaccurate, in our case clock skew was about 30-40 minutes per one day. To check the time skew when running tests, we used NTP time synchronization and checked that the time offset is not too large.

4. PERFORMANCE EVALUATION AND DISCUSSION

We tried to cover a wide range of HPC applications using HPCC, NPB and SPEC MPI2007 benchmarks in different configurations. The following subsections describe our experimental setup, testing methodology and results. At the end of this section, we analyze and discuss the reasons for performance overhead in different cases.

Experimental setup

We use two HP clusters as a testbed. First cluster consists of 8 HP ProLiant BL2x220c G7 blades. For tests we used up to 8 nodes. Each node contains 2 Intel Xeon X5670 CPUs (6 cores per CPU) and 24GB of RAM. Hyperthreading was disabled on all nodes. For running SPEC MPI2007 we used second cluster which consists of 20 HP ProLiant SL390s G7 nodes, each containing 2 Intel Xeon X5650 CPUs (6 cores per CPU) and 24GB of RAM. Cluster nodes communicate via 1Gbit/s. service Ethernet network, and 40Gbit/s. Infiniband network used for computing.

The systems used for native tests, as a host OS for KVM and as a guest OS are Linux CentOS 6.0-6.3. The kernel is modified to support Palacios device assignment. MPI library is OpenMPI 1.4.3, GCC version used to build test suites is 4.4.4. Infiniband stack is provided by Mellanox OFED for Linux, version `1.5.3-1.0.0-rhel6-x86_64.` QEMU version is `qemu-kvm-1.0.1` with our modifications. The Kitten OS version is 1.2.0 with modifications required to support device assignment in Palacios' guests. The guest system is configured with 16GB of RAM, 12 processor cores, with or without NUMA architecture emulation.

[1]We contributed our code to the V3VEE project, some of our patches could be found at the V3VEE project Web site [3] and some are already integrated into the development branch.

Benchmarks.

To measure the performance we use HPC Challenge, NAS Parallel Benchmark suites (first cluster, running on 2, 4 and 8 nodes) and SPEC MPI2007 suite (second cluster, running on 4, 8, 16, 20 nodes).

NPB version is 3.3.1 for MPI (NPB3.3-MPI). From NPB suite we employ IS, EP, CG, MG, FT and LU tests with the problem class C. IS stands for Integer Sorting. EP is Embarrassingly Parallel Gaussian random variates generator. CG is a program using Conjugate Gradient method. MG (MultiGrid) approximates the solution to discrete Poisson equation. FT is discrete 3D fast Fourier Transform with all-to-all communication. LU is a solver for systems of partial differential equations. We run 8, 9 or 12 processes per node depending on test constraints. Results are expressed in seconds.

HPCC version is 1.4.1 with ATLAS 3.8.4. Tests from HPCC suite are executed using 12 processes per node. We assess the results of STREAM, RandomAccess and HPL tests from HPCC package. STREAM measures sustainable memory bandwidth. We use data from Single (single process is computing) and EP (Embarrassingly Parallel — all processes compute the same thing independently) versions of this test when running on one node. RandomAccess measures the rate of integer updates to random memory areas in GigaUpdates per second (GUP/s). This test is known to be very challenging for virtualization since it provides huge TLB pressure. HPL measures the rate of solving for linear system of equations. Results are provided in GFlops. For HPL we use the following parameters: matrix size is 30000, block size is 150, computation grid is square when it is possible. We exclude other HPCC benchmarks' results due to the space limitations.

SPEC MPI2007 version is 2.0. Currently we gathered data on native system and KVM system with NUMA emulation. We selected GemsFDTD, lammps, pop2, socorro, tachyon, wrf2, zeusmp2 medium size tests from SPEC MPI2007 package. Tests description is available at SPEC website [2].

Virtual machines are known to have problems with timing. Hypervisor and host OS provide additional noise source and cause guest timers to be less precise. As a result, run times in virtual environment tend to be more scattered. To make our results more precise, we perform most of NPB runs 50 times, HPCC runs 20 times and SPEC runs 10 times. We report the estimate of the mean together with 97% confidence interval for the mean value calculated using Student's t distribution to understand the reliability of gathered data. Though consequent test runs cannot be independent, confidence interval could help us to estimate the influence of different measurement error sources and to be sure that we did enough consequent runs.

Results

We started our testing with the "default" QEMU configuration, using default memory allocation mechanism. Also initially we tried virtual CPU pinning to prevent QEMU threads migration, and checked that HugeTLBfs mechanism provides the same overhead as Transparent Hugepages. Results for all these configurations were almost the same, with up to 50% performance overhead in some cases. After that

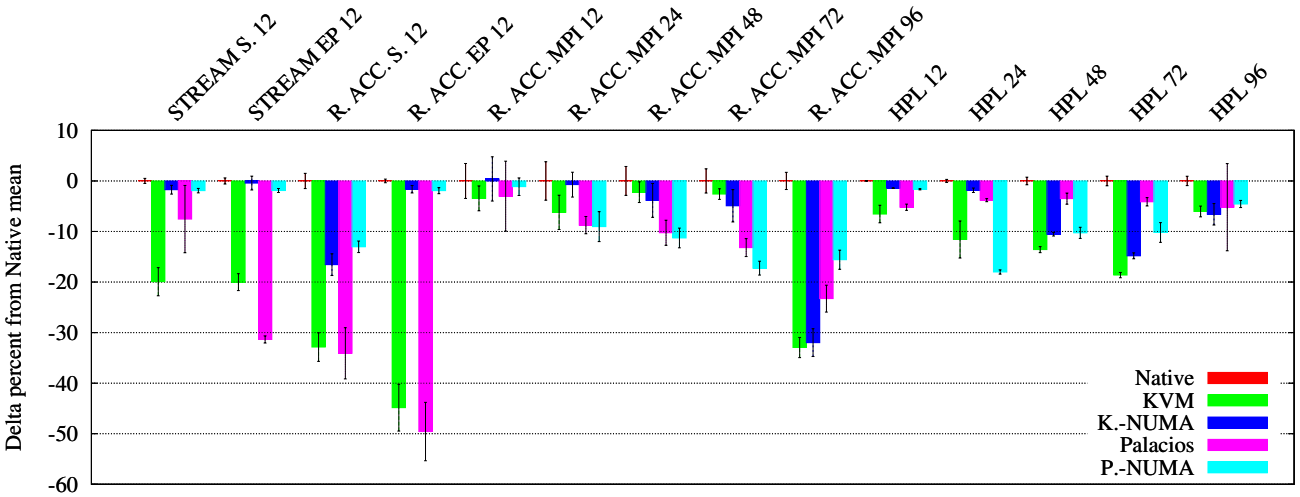


Figure 3: KVM/QEMU and Palacios results for HPCC suite.

we figured out that the reason for most overhead is the NUMA architecture of our test hardware. QEMU emulates an SMP system by default, which causes performance penalties when some virtual CPU accesses non-local memory. The next step was to provide the NUMA architecture to virtual machines corresponding to the real NUMA configuration. It was performed using QEMU NUMA emulation support, as described in the previous section.

For Kitten/Palacios system we tested two configurations – with and without NUMA support. In both cases 2MB nested pages were used for the guest memory. Overall NPB results comparing configurations of KVM/QEMU and Palacios are provided in figures 1, 2 with error bars denoting 97% confidence interval for each test mean. Data is plotted in relation to the Native case which has the value of zero in each bar group. Key under each bar group consists of the test name and process count used for computation. Results for HPCC suite are in Figure 3. "R. ACC." stands for Random Access, "S." is a Single version of the test.

The first conclusion for NPB suite results is that NUMA-awareness matters. Without NUMA Palacios and KVM show more than 60% and 40% overheads on MG 8 test, but with NUMA this overhead disappears. The same situation is true for the most other tests. Another fact concerning NUMA is that for almost all tests without NUMA, overhead decreases while the number of processes grows (the difference for one test could be 30% and more, see IS and MG tests). At the same time, for NUMA case in many tests overhead varies slightly.

For HPCC tests, NUMA emulation improves performance results for STREAM and Random Access Single and EP versions. For Random Access EP, NUMA-enabled guest almost eliminates 45-50% overhead of the non-NUMA guest. For MPI Random Access, the situation is not so definite – NUMA-enabled guest may perform better or worse.

The next thing we can conclude is that Palacios implementation has some drawbacks when compared to KVM – it can be seen when comparing results without NUMA support for almost all NPB tests and HPCC Random Access test with 24, 48 and 72 processes. The maximum difference is about 24% on LU 12 test. Also Palacios shows strange results for EP 96 test. HPL results are strange too: on 24, 48 and 72 processes Palacios without NUMA emulation shows the best result, but with NUMA emulation enabled overhead increases a few times. Our hypothesis is that KVM/QEMU has an important advantage: Linux host kernel automatically arranges memory and CPUs in the "default" QEMU configuration, while Palacios's VM has predefined virtual CPU to physical core mapping and memory ranges.

We can also see that sometimes the results inside VM with NUMA support are better than native. For example, look at tests CG, MG, MG, where KVM behaves better than native. Probably the reason for this behavior is that some unaccounted factors were present during these runs.

Finally, the CG, FT and IS test results with 64 processes and Random Access MPI results with 96 processes show the advantage of Kitten with Palacios over KVM/QEMU in the NUMA case. This advantage ranges from 3.5 to 21%. Probably this advantage is associated with the decreased noise of Kitten OS when compared to Linux, especially for fine-grained IS test. In general, we can conclude that KVM provides more stable and predictable results, while Palacios is better on fine-grained tests. Meanwhile, Palacios shows abnormal performance degradation on some tests.

SPEC MPI2007 results.

Our current SPEC MPI2007 results are available only for KVM with NUMA emulation enabled with up to 240 cores used for computation. The results are presented in Figure 4. We can see that GemsFDTD, lammps, tachyon and wrf2 scale well with persistent overhead around 5-10%. At the same time, other three tests show noticeable overhead ranging from 20 to 43%. In fact, we see the clear division of applications into two classes; among these seven tests, more than a half can be virtualized successfully. For the rest tests, the reasons for overhead should be thoroughly investigated.

5. CONCLUSION

Our primary contribution has been to demonstrate the importance of proper hypervisor and host OS tuning when running HPC task inside virtual machines, including NUMA architecture emulation according to the real configuration. In particular, we explored KVM/QEMU and Palacios hypervisors. KVM is widely used in industry while Palacios is

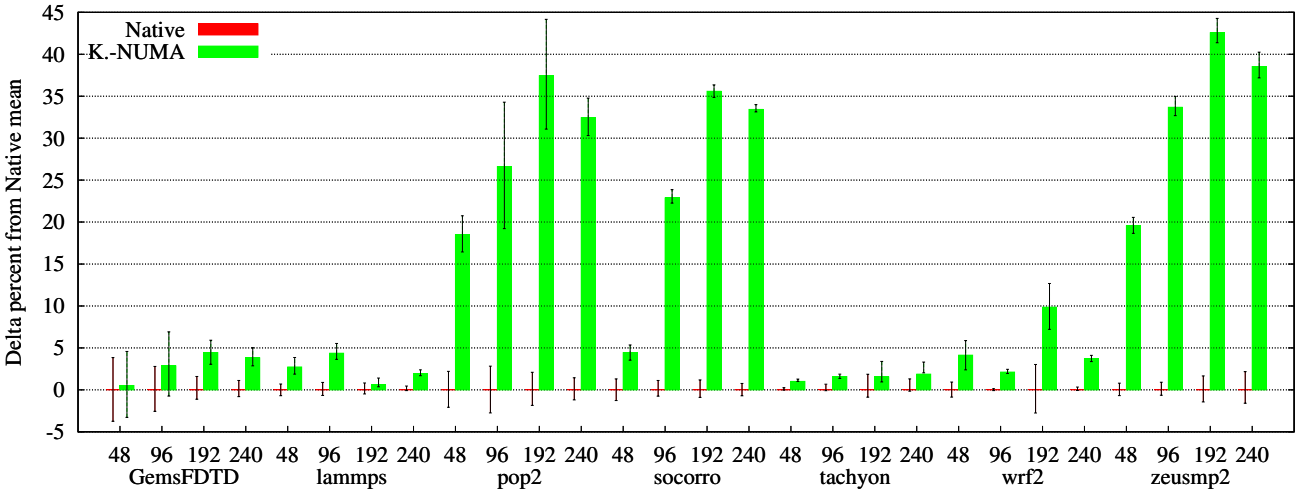


Figure 4: KVM/QEMU results for SPEC MPI2007 suite.

a young project targeted at HPC virtualization. We patched QEMU and Palacios to make the real NUMA topology available in the guest system. By using proper NUMA emulation, we reduced the performance degradation from 10-60% to 1-5% on many tests from HPCC and NPB suites. At the same time, SPEC MPI2007 results show that some applications still suffer from overhead ranging from 20% to 45%, while other scale well with core count up to 240.

We assess KVM/QEMU and Palacios hypervisors and conclude that in general their results with NUMA emulation enabled are similar, with KVM providing more stable and predictable results and Palacios being much better on fine-grained tests, but showing abnormal performance degradation on some other tests. The noise of the host OS is really important for fine-grained tests scalability and in this respect Kitten behaves better than Linux resulting in better scaling for tests running inside Palacios' virtual machines. We believe that the noise amount generated by virtualization system will become crucial for successfull virtualization of large-scale HPC systems.

6. REFERENCES

[1] OpenVZ: container-based virtualization for Linux, http://openvz.org/.

[2] SPEC MPI2007 Documentation, http://www.spec.org/mpi/Docs/.

[3] V3VEE: An Open Source Virtual Machine Monitor Framework For Modern Architectures, http://v3vee.org/.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center. December 1995.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, Oct. 2003.

[6] K. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008.*, pages 1 –12, November 2008.

[7] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. Abstract High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt), in conjunction with EuroSys 2007*, 2007.

[8] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, 2012 (to appear).

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[10] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1 –12, April 2010.

[11] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 169–180, New York, NY, USA, 2011. ACM.

[12] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[13] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder. SPEC MPI2007 – an application benchmark suite for parallel systems using MPI. *Concurr. Comput. : Pract. Exper.*, 22(2):191–205, Feb. 2010.

[14] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *2003 ACM/IEEE Conference on Supercomputing*, page 55, November 2003.

[15] N. Regola and J.-C. Ducom. Recommendations for virtualization technologies in high performance computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 409–416, Washington, DC, USA, 2010. IEEE Computer Society.

[16] J. Watson. VirtualBox: bits and bytes masquerading as machines. *Linux J.*, Feb. 2008.

[17] A. J. Younge, R. Henschel, J. Brown, G. von Laszewski, J. Qiu, and G. C. Fox. Analysis of Virtualization Technologies for High Performance Computing Environments. In *The 4th International Conference on Cloud Computing (IEEE CLOUD 2011)*, July 2011.

GENICloud and TransCloud: Towards a Standard Interface for Cloud Federates

Andy Bavier
Princeton University
acb@cs.princeton.edu

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

Tony Mack
Princeton University
tony.mack@gmail.com

Chris Matthews
University of Victoria
cmatthew@cs.uvic.ca

Joe Mambretti
Northwestern University
j-mambretti@northwestern.edu

Rick McGeer
Hewlett-Packard Laboratories
rick.mcgeer@hp.com

Paul Mueller
TU-Kaiserslautern
pmueller@informatik.uni-kl.de

Alex Snoeren UCSD
snoeren@csucsd.edu

Marco Yuen
Princeton University
marcoy@gmail.com

ABSTRACT

In this paper, we argue that federation of cloud systems requires a standard API for users to create, manage, and destroy virtual objects, and a standard naming scheme for virtual objects. We introduce an existing API for this purpose, the Slice-Based Federation Architecture, and demonstrate that it can be implemented on a number of existing cloud management systems. We introduce a simple naming scheme for virtual objects, and discuss its implementation.

1. Introduction

"Cloud" systems and services refer to the remote allocation and use of various virtual resources over the Internet. These resources can range from virtual machines, block stores, and databases (Amazon EC2 and S3, Rackspace, HP Cloud), to threads (Microsoft Azure) to language engines and VMs (Google App Engine). OpenCirrus[1, 20] offers a federated collection of local clusters, each of which offer a number of virtual resources. The US Global Environment for Network Innovations[12] project offers a heterogeneous collection of virtual networked resources for researchers in networking, distributed, and cloud systems.

One model for federated systems, the so-called "Cloud-bursting" model, involves the use of a relatively small number of homogeneous systems. In this model, a secondary cloud system is used for reasons of capacity or cost when the primary cloud system is saturated. A second model, which is of interest to us, is when a large number of potentially heterogeneous resources are allocated from a large number of different systems, from different administrative domains. There are a variety of use cases which motivate this second, "Ubiquitous Cloud" model. Content distribution systems which require proximity to widely-distributed users; robust stores for critical data, dispersed to prevent damage or destruction in the event of local catastrophes; wide-area sensing systems with heavy-computation based back ends; experimentation on wide area distributed systems or cloud infrastructure systems; and so on.

An immediate and compelling need for the Ubiquitous Cloud model is the need to create large-scale testbeds for the research and education community. Industrial practice has moved far beyond the capacity of academic research. A Yahoo! "clique" of servers, an internal unit of management, comprises 20,000 servers and perhaps a quarter-million cores. No single academic testbed has anything like that capacity.

The Ubiquitous Cloud model exposes a number of issues in scalability of cloud systems. The most prominent of these is the need for a common API across heterogeneous systems. Large-scale distributed systems will typically not be instantiated or managed by a Graphical User Interface, but by tools. Indeed, users today largely instantiate Cloud jobs on Eucalyptus[9], OpenStack[21] or some commercial systems using the popular command-line euca2ools[8]. The euca2ools have hard-coded the API to Eucalyptus systems, which was designed to be API-compatible with the popular commercial offering Amazon EC2; OpenStack, a later, open-source entrant, supports this API.

Though the Amazon API is emerging as a de facto standard in this area, it has a number of weaknesses as a general solution. The primary weakness is that it is tuned specifically for virtual machine based compute resources and block stores, and so it has spread only across systems that offer roughly the same set of services that Amazon does. Allocation of resources across heterogeneous systems (such as, for example, allocating a collection of Restricted Python[7] processes from the Seattle[6, 18, 30] testbed, a collection of bare hardware nodes from ProtoGENI[28], some cluster-based compute nodes from GENI-Cloud[16], and some wireless or wimax nodes will require an API that permits the user to specify, and the cloud systems to advertise, not simply the number but the *kind* of available resources.

Authorization and authentication present significant challenges. Different organizations control different facilities, and operate in different economic and legal environments. It is impractical to expect that these organizations will concur on a common user database, common set of roles, common passwords, or common Acceptable Use Policies. Federated authentication solutions such as Shibboleth[32] have been used, but

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, September 21, 2012, San Jose, CA, USA.

present some logistical difficulties: in particular, Shibboleth assumes a persistent and reliable connection between the organization providing the facility and the organization certifying the user, and a single oracle for any particular user. Again, this has been shown to work for a collection of relatively homogenous environments, but presents obvious problems when used for a heterogeneous set of facilities.

In this paper, we discuss the use of the GENI standard Slice-Based Federation Architecture[25] as a standard API for heterogeneous cloud systems. The SFA incorporates a simple API which fits well on existing cloud systems, and augments this with a resource-specification standard for both requests and allocation, the RSpec. It further incorporates an authorization scheme based on unforgeable, self-validating certificates, eliminating the need for a centralized database, clearinghouse, or federated identity mechanism. We show that it has been used succesfully on a wide variety of virtual resource-as-a-service facilities.

Our contribution in this paper is to describe an implementation strategy for the SFA onto legacy Cloud systems. We have successfully used an SFA implementation to control both OpenStack and Eucalyptus-based clusters, and use it today to offer seamless federation between a PlanetLab-based infrastructure and an OpenStack-based infrastructure.

Further, when a user creates a large collection of virtual resources, he must be able to name and access them predictably and easily without manually naming each one. We propose a simple URL-based naming scheme for these resources.

The remainder of this paper is organized as follows. In Section 2, we give a high-level overview of the SFA, with some short examples of its use. In Section 3, we describe a simple implementation strategy for the SFA on legacy cloud systems that we have used succesfully on OpenStack and Eucalyptus. In Section 4 we describe TransCloud, a naming scheme for large, heterogeneous, multi-site slices of clouds. In Section 5 we draw some conclusions, and offer suggestions for future work.

2. The Slice-Based Federation Architecture

The Global Environment for Network Innovations (GENI) program is a US National Science Foundation program to construct a next-generation testbed for networking and distributed systems research. It was designed by a team of over 50 US systems researchers in 2006-2007, following the success of two precursor systems: Emulab[10, 35] a hardware-as-a-service cluster at the University of Utah devoted to network experiments and emulation, and PlanetLab[4, 24, 27, 33], a worldwide containers-as-a-service distributed systems platform developed by Princeton University, UC-Berkeley, the University of Washington, Intel, and HP.

GENI soon grew to encompass a number of other control frameworks and testbeds:

1. ORBIT[22], an experimental testbed for wireless nodes from the University of Rutgers

2. Seattle[30], a high-level distributed Python environment from the University of Washington and NYU-Poly

3. FOAM[11], a programmable-networks-as-a-service system from Stanford, UC-Berkeley and BigSwitch systems, which offers isolated layer-2 networks based on OpenFlow[17, 19] and FlowVisor[31] technology.

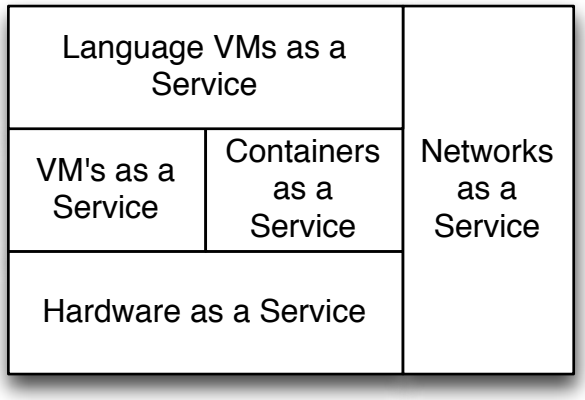


Figure 1: A Simplified Form of the As A Service Stack

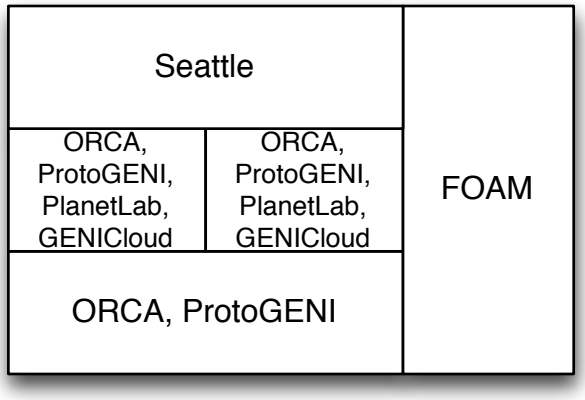


Figure 2: A Simplified Form of the GENI Stack

4. ORCA[2, 3], a substrate management system from Duke University offering multiple physical and virtual compute resources.

A simplified version of the various computing elements offered as services is shown in Figure 1. Most of these elements are available from commercial providers. Google App Engine, for example, is an excellent example of language VM's as a stack.

The stack is both more complex than is shown in Figure 1 and contains more elements; we have simplified it for explanatory purposes. For example, Language VM's can rest directly on hardware, and need not be mediated through virtual machines or containers, and containers can rest on VM's if desired. Further, Storage as a Service (such as Amazon S3) is another vertical pillar. Nonetheless, this serves to illustrate the point: distributed systems can be built from a combination of virtual resources of various types, and an API must accomodate those.

The GENI frameworks fit into the services stack in Figure 1, with the mapping shown in Figure 2. Again, this is a simplified picture; it omits ORBIT, and has the same simplifications as in Figure 1 .

GENI experiments could be expected to span any subset of these control frameworks. Moreover, each of these was a codebase, which could be expected to be replicated over a number of administrative domains. Large experiments are expected to require the marshalling and orchestration of large numbers of

heterogeneous resources. In sum, software tools were expected to create and manage disparate resources spread over multiple administrative domains, running under different management suites. A common API spanning all of these control frameworks was required, with a common authorization method and a common method of advertising resources and servicing resource requests. The common API is the *Slice-based Federation Architecture*. We detail it here.

The central abstraction in the Slice-Based Federation Architecture is a *slice* of the underlying physical substrate. A slice refers to the entire collection of virtual compute, communication, and storage resources devoted to an experiment. One can think of it as a virtual, distributed network of virtual machines devoted to a specific project. For example, the *CoDeeN*[23, 34] slice on PlanetLab consists of several hundred Linux VServers[15], distributed at several hundred sites worldwide, which together implement a Content Distribution Network. The GENIS3Monitor slice[5] is a scalable, extensible, and safe network monitoring system for research networks and clouds.

The notion of a slice is not common among Cloud management frameworks, but it provides a number of services to end-hosts and users. For example, it makes possible global manipulation of the elements of a slice as a unit, without separate operations on each individual element. Good examples of this are loading user certificates onto every individual element of the slice, or doing software loads and boot scripts on every element of the slice. It also provides a natural unit of resource allocation and accounting.

Individual virtual resources within slices are referred to as *slivers*. Again, a sliver is simply a generalization of the concept of a virtual machine to encompass containers, physical machines, or even individual named block stores.

The API exported by the SFA is similar to that exported under the euca2ools, or the OpenStack API, with an additional overlaying set of APIs to manage sets of resources, or slices. It is an XML/RPC interface over HTTPS, with authentication by credential.

The central data structure in the SFA is the Resource Specification, or RSpec[29]. It is GENI's mechanism for advertising, requesting, and describing the resources used by a slice. In general, it is an XML document. For a machine it will often describe the instance size requested, whether it needs exclusive access to a physical host (i.e., requires a host or can live with a VM), may describe a particular image to be loaded, and so on. We show one simple example abstracted from the ProtoGENI RSpec examples, with various namespace details omitted for space in clarity in Figure 3. The full example can be found at: `http://www.protogeni.net/trac/protogeni/wiki/RSpecRequestDiskImageExample`

The RSpec example shown in Figure 3 requests a physical host ("raw-pc") and a Fedora Core 10 OS. This is an example of a *request* RSpec, which would be used by a tool to request a physical node with a specific OS. In addition, an *advertisement* RSpec is used by a manager of physical resources to show what is available. A simplified example is shown in Figure 4, taken from `http://www.protogeni.net/trac/protogeni/wiki/RSpecAdSingleNodeExample`

This particular RSpec describes a node with name "pc160", which is of type "pc850" (850 MHz x86 processor), currently available for exclusive use. It also matches requests for "pc" (any x86-based processor). It has two network interfaces.

```
<rspec type="request">
  <node client_id="exclusive-0"
      component_manager_id="
      urn:publicid:IDN+emulab.net+
      authority+cm" exclusive="true">
    <sliver_type name="raw-pc">
      <disk_image name="urn:publicid:IDN+
          emulab.net+image+emulab-ops//
          FEDORA10-STD"/>
    </sliver_type>
  </node>
</rspec>
```

Figure 3: ProtoGENI Request RSpec Example

```
<rspec    type="advertisement"
    generated="2009-07-21T19:19:06Z"
    valid_until="2009-07-21T19:19:06Z" >
  <node component_manager_uuid="
      urn:publicid:IDN+emulab.geni.emulab.
      net+authority+cm"
        component_name="pc160"
        component_uuid="urn:publicid:IDN+
            emulab.geni.emulab.net+node+
            pc160" >
    <node_type type_name="pc850"
        type_slots="1" />
    <node_type type_name="pc" type_slots=
        "1" />
    <available>true</available>
    <exclusive>true</exclusive>
    <interface component_id="
        urn:publicid:IDN+emulab.geni.
        emulab.net+interface+pc160:eth0"/>
    <interface component_id="
        urn:publicid:IDN+emulab.geni.
        emulab.net+interface+pc160:eth1"/>
  </node>
</rspec>
```

Figure 4: ProtoGENI Advertisement RSpec Example

In addition (for reasons of space we will omit an example) there is a third type of RSpec: the Manifest Rspec. This describes a resource requested by and granted to a slice; it shows a fulfilled request. Though the example RSpec's we have shown have come from ProtoGENI, the essential features are uniform across the various GENI Control Frameworks.

Physical devices in the SFA are referred to as *Components*, which are formally defined as collections of physical resources. A simple example is a server, but others include programmable switch, Open WRT access point, dedicated software router, and so on. Components are controlled by *Component Managers* (note that in Figure 4, the node rspec had a component_manager attribute; that had the URN of manager of that component). One can think of a Component Manager as having roughly the same capabilities and duties as a Eucalyptus Node Controller.

Components in the SFA are grouped into *Aggregates*. Again, a simple example of an Aggregate is a cluster; another exam-

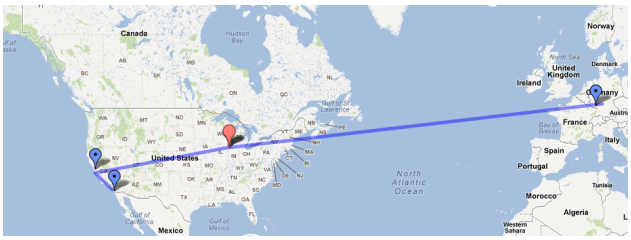


Figure 5: GENICloud Sites

ple is a distributed platform such as PlanetLab, or a collection of OpenFlow switches. An Aggregate is managed by an *Aggregate Manager*, which has roughly the same function as a Eucalyptus Cluster Controller. However, Aggregates can be hierarchical, which means that Aggregates can contain other Aggregates as well as Components. One use case for this is where a cluster such as ProtoGENI contains a set of OpenFlow switches with a distinct manager; in order to allocate a slice containing dedicated nodes hooked up to a progammable network slice, the cluster AM makes requests of the AM responsible for the collection of OpenFlow switches. This implies that the AM interface is a superset of the CM interface, since the AM must respond to component requests.

The SFA defines global identifiers (GID) for all entities in the federated system: principals, slices, physical components, services, etc.[25] The GID is an unforgeable certificate, signed by one or more entities, that binds together three pieces of information: the object's public key, a unique identifier (e.g., a UUID), and a lifetime. Any entity may verify a GID via cryptographic keys that lead back, through a chain of endorsements, to a well-known root; in this way the receiver of a GID can authenticate that the sending object is the one to which the GID was actually issued. A GID signed with its own private key is called a self-certifiying ID or SCID. A SCID establishes the issuing entity's right to assert attributes or authorization rules for the named object.

3. Implementing the SFA on Legacy Cloud Management Systems

The SFA has proven to be an effective abstraction over existing legacy infrastructures such as Emulab and PlanetLab, and thus a candidate as an effective overlying API for general as-a-Service computational infrastructures. Further, it has been able to accomodate unanticipated infrastructures such as virtualizable switching fabrics (OpenFlow).

We have been investigating whether the SFA can be used as an overlying API for general cloud frameworks in the GENICloud[16] project. We sought to answer three questions:

1. Can the SFA overlay an existing cloud management system, designed entirely independently of the SFA?
2. Can we manage clusters using *different* specific management systems, each of which presents an SFA interface, and create slices which span the different systems?
3. Can we use the SFA to manage a multi-site cloud system?

We built a four-site Cloud system, using varying underlying resources to manage each site, and managing them with the SFA. We show the sites in Figure 5. The four are HP Labs, as part of the OpenCirrus cluster[20], Northwestern University, UC San Diego, and the Technical University of Kaiserslautern.

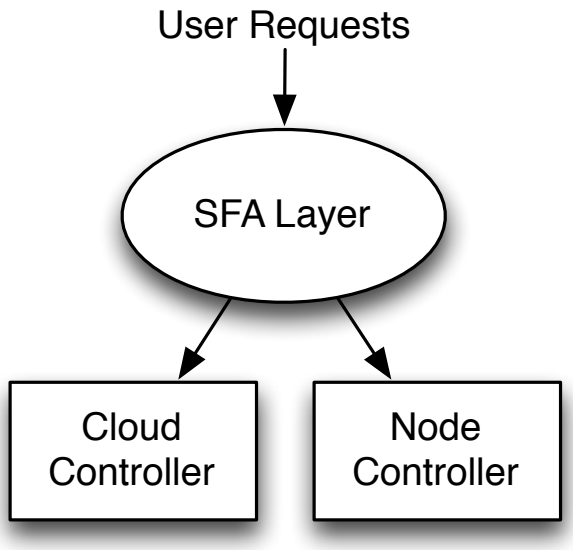


Figure 6: SFA-on-Cloud Architecture

The four sites run different management stacks on the local clusters, but appear homogeneous through the SFA.

A further motivation for the GENICloud effort was to federate cloud systems into the distributed and experimental infrastructures provided by GENI. Current GENI platforms largely are not designed to scale rapidly on demand. Under the federation of a standard Cloud platform and GENI, a more comprehensive platform is available to users; for example, development, computation, data generation can be done within the cloud, and deployment of the applications and services can be done on distributed platforms (e.g., PlanetLab). By taking advantage of cloud computing, GENI users can dynamically scale their services on GENI depending on the demand of their services, and benefit from other services and uses of the cloud. Take a service that analyzes traffic data as an example; the service can deploy traffic collectors to collect Internet traffic data on PlanetLab . The traffic collected can be stored and processed in the cloud.

PlanetLab, being a part of the GENI project as one of the control frameworks, has high global penetration. However, it was not designed for either scalable computation nor large data services. Some services and experiments require a huge amount of data or they need to persist a large amount of instrumental data; also, it lacks the computation power for CPU intensive services or experiments. GENICloud fills in the gap by federating heterogeneous resources, in this case, a cloud platform with PlanetLab.

Most of the implementation effort of GENICloud concentrated on implementing the aggregate manager on top of the existing Cloud Controller, under both Eucalyptus and OpenStack. A key design choice was whether to modify the existing Controller, or use an overlying controller which implemented SFA API calls by calling through to the underlying controller.

We chose the latter, and the resulting architecture is shown in Figure 6. In this architecture, both the fact that the user is coming through an SFA layer and the identity of the underlying controller is hidden. The SFA aggregate manager acts an mediator between user requests and an underlying cloud. The primary advantage is that we need not maintain modifications

and patches in existing Cloud managers, updating them as new modifications come out; rather, we only need up date our Aggregate Manager as the interface to the Cloud manager changes. We have not yet implemented, but do not exclude, Cloud controller specific optimizations for controllers with specific optimizations such as Tashi[13]. A secondary advantage is that the identity of the underlying Cloud controller is hidden from the user, meaning that user-facing tools and scripts don't change when the underlying Cloud controller is hidden.

The Aggregate Manager manages the creation of cloud instances for a slice, and maintains a mapping of slices and instances. The Aggregate Manager offers RSpecs for the VM's and containers created by the underlying cloud, and manipulates the underlying Node Controller and Cluster Controller to serbvice user requests.

The resource specification format for VM's and containers closely mirrors the RSpec formats seen previously, and indeed we were able to re-use the ProtoGENI RSpecs for Virtual machines, and the PlanetLab RSpecs for containers.

In our design, the SFA is essentially a proxy between the user and OpenStack. Users authenticate to the SFA and are authorized to perform specific actions in the SFA API using SFA credentials. All authorization policy checks occur in the SFA layer. This insulates the Cloud Controller from any overlying policy changes at the SFA layer, such as the proposed move towards Attribute-Based Access Control, or ABAC, a self-verifying capability system[14].

We have implemented the SFA layer over OpenStack and Eucalyptus. Currently, our Palo Alto site is running OpenStack under the SFA, and UCSD, Northwestern and TU-Kaiserslautern are running PlanetLab under the SFA, using the MyPLC cluster controller[26]. We have succesfully transitioned the Palo Alto site from Eucalyptus to MyPLC to OpenStack, without changing the overlying SFA interface.

4. Naming Scheme: TransCloud

A key for large-scale, multi-site, multi-administrative domain slice is the ability for the user to administer his slice, orchestrate the action of the slice's slivers, and easily configure the slivers in his slice to communicate. In order to do this, a standard naming scheme for slivers is required. DNS services will take care of communication when the slice is set up.

Here, we propose such a naming scheme and DNS implementation. Our scheme is inspired by the naming scheme of Emulab[10]. In Emulab, the fundamental subdomain is a "project", which is a group of researchers who create "experiments" in pursuit of some research goal. Emulab's experiment is simply a slice; a project is an overlying space used for resource allocation, directory structures, storage of related data, and also provides a namespace to permit researchers to isolate namespaces.

Using the terminology that we've been using in this paper, each sliver in an Emulab slice is named `<sliver-name>.<slice-name>.<project-name>.emulab.net`; e.g., `sender.baselineqos19.chart.emulab.net`. In this case, `sender` names the node, `baselineqos19` the experiment, and `chart` the project. Nodes within a slice can simply use the `<sliver-name>` and this resolves to the appropriate FQDN. Use of project as a namespace partition is not in the SFA and is not a feature of other control frameworks such as PlanetLab, but it has its uses. In particular, PlanetLab prepends the name of the host institution on a slice name to prevent slice namespace contention; the CoDeen slice is not`codeen`, but rather `princeton_codeen`. The PlanetLab convention is an alternative.

Emulab achieves this by using its own internal DNS servers and updating the DNS entries when an experiment is swapped in. We propose to use the same method.

Emulab was conceived as a single-site, single-administrative domain testbed. We are designing for multi-site, multi-domain administration, so we modify the Emulab scheme to accomodate this. This means that we must incorporate both the site name and the administrative domain name in the URL; both must be accomodated because the *site* of a sliver can be important (consider a Content Distribution Network or real-time sensitive server, for example); and the administrative domain is the entity that will actually allocate the slice.

There are two fundamental principles that underlie our scheme. First, administrative domains must make only three agreements: they will run the SFA, accept GID's as credentials, and implement the TransCloud naming schement. The second principle is that names should be allocated to slices and projects at highest level in the hierarchy possible.

The first principle argues that projects and slices do not span administrative domains, since this would imply that administrative domains need to agree on a common project namespace. The second argues that slices and projects should be defined at the administrative domain level, since a single administrative domain can administer projects and slices that span multiple sites, and there is value to users in creating multi-site slices.

We have obtained `trans-cloud.net` as the root domain, so our scheme is then: `<sliver-name>.<site-name>.<slice-name>.<project-name>.<domain-name>.trans-cloud.net`. Again, appropriate DNS work will permit the use of `<sliver-name>.<site-name>` within a slice.

Root storage at a site, and global ssh logins, should be to `users.<site-name>.<domain-name>.trans-cloud.net`; this is the equivalent of `users.emulab.net`.

Interactions with the Domain authority occurs through `<authority-name>.trans-cloud.org`; the `.org` domain is for administrative access (e.g., certificate upload and renewal, slice creation and management, and the like); the `.net` domain for experimental access.

5. Conclusions

In this paper we have demonstrated the implementation of SFA on two legacy Cloud architectures, and shown the feasibility of a cross-site, cross-manager SFA implementation. GENI-Cloud is a single administrative domain spanning four sites and two aggregate managers. We have proposed a naming scheme for a unified, multi-site, multi-domain distributed cloud infrastructure.

Acknowledgements

This work was partially supported by the GENI Project Office under contract GENI 1779B. The GENI Project Office is funded by the National Science Foundation's CISE Division.

6. References

[1] A. Avetisyan, R. Campbell, I. Gupta, M. Heath, S. Ko, G. Ganger, M. Kozuch, D. O'Hallaron, M. Kunze, T. Kwan, K. Lai, M. Lyons, D. Milojicic, H. Y. Lee, Y. C.

Soh, N. K. Ming, J.-Y. Luke, and H. Namgoong. Open cirrus: A global cloud computing testbed. *Computer*, 43(4):35 –43, april 2010.

[2] I. Baldine, Y. Xin, A. M, C. Heermann, J. Chase, V. Marupadi, A. Yumerefendi, and D. Irwin. Networked cloud orchestration: A geni perspective. In *2010 Globecom Workshops*, 2010.

[3] I. Baldine, Y. Xin, A. Mandal, P. Ruth, A. Yumerefendi, and J. Chase. Exogeni: A multi-domain infrastructure-as-a-service testbed. In *Proceedings Tridentcom, 2012*, 2012.

[4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI*, May 2004.

[5] E. Blanton, S. Chatterjee, S. Gangam, S. Kala, D. Sharma, S. Fahmy, and P. Sharma. Design and evaluation of the s^3 monitor network measurement service on geni. In *COMSNETS*, pages 1–10, 2012.

[6] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. *SIGCSE Bull.*, 41(1):111–115, 2009.

[7] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code. In *The 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, 2010.

[8] Euca2ools. `http://open.eucalyptus.com/wiki/Euca2oolsGuide_v1.3`.

[9] Eucalyptus. `http://www.eucalyptus.com/`.

[10] Emulab website. `http://www.emulab.net`.

[11] Foam. `https://openflow.stanford.edu/display/FOAM/Home`.

[12] Geni project website. `http://www.geni.net`.

[13] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. López, M. Stroucken, and G. R. Ganger. Tashi: location-aware cluster management. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 43–48, New York, NY, USA, 2009. ACM.

[14] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[15] Linux vservers. `http://linux-vserver.org/Welcome_to_Linux-VServer.org`.

[16] R. McGeer, A. AuYoung, A. Bavier, J. Blaine, Y. Coady, J. Mambretti, C. Matthews, C. Pearson, A. Snoeren, and M. Yuen. Transcloud: Design considerations for a high-performance cloud architecture across multiple administrative domains. In *Proceedings CLOSER*, 2011.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev*, 38:69–74, March.

[18] Monzur Muhammad and Justin Cappos. Towards a Representive Testbed: Harnessing Volunteers for Networks Research. In *The First GENI Research and Educational Workshop*, GENI'12, 2012.

[19] Openflow website. `http://www.openflow.org`.

[20] Opencirrus. `http://opencirrus.org/`.

[21] Openstack. `http://openstack.org/`.

[22] Orbit. `http://www.winlab.rutgers.edu/docs/focus/ORBIT.html`.

[23] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets*, 2003.

[24] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences implementing planetlab. In *OSDI*, November 2006.

[25] L. Peterson et al. Slice-based federation architecture, v 2.0. `http://groups.geni.net/geni/attachment/wiki/SliceFedArch/SFA2.0.pdf`.

[26] MyPLC User guide. `http://www.planet-lab.org/doc/myplc`.

[27] Planetlab. `http://www.planet-lab.net`.

[28] Protogeni web site. `http://www.protogeni.net/trac/protogeni`.

[29] Protogeni rspec. `http://www.protogeni.net/trac/protogeni/wiki/RSpec`.

[30] Seattle. `https://seattle.cs.washington.edu/`.

[31] R. Sherwood. Safely using your production network as a testbed. *Login; Magazine*, 36(1), February 2011.

[32] Shibboleth. `http://shibboleth.net`.

[33] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using planetlab for network research: myths, realities, and best practices. In *IN PROCEEDINGS OF THE SECOND USENIX WORKSHOP ON REAL, LARGE DISTRIBUTED SYSTEMS (WORLDS)*, pages 17–24, 2006.

[34] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the codeen content distribution network. In *In USENIX Annual Technical Conference, General Track (2004*, pages 171–184, 2004.

[35] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and network. In *Proceedings of OSDI 02*, 2002.

A Mechanism to Measure Quality-of-Service in a Federated Cloud Environment

Shoumen Bardhan
Hewlett-Packard Enterprise Services
4000 N Mingo Road
Tulsa, OK 74116
918.625.1833
shoumen.bardhan@hp.com

Dejan Milojicic
Hewlett-Packard Labs
1501 Page Mill Rd
Palo Alto, CA 94304
650.236.2906
dejan.milojicic@hp.com

ABSTRACT

In a federated Cloud environment, services may be composed of other services from different Clouds with different Cloud provider Quality-of-Service (QoS) guarantees. Providers running services on the multiple Clouds will be contractually obligated to meet or exceed the QoS which they have agreed to provide to their consumers. A key challenge for the service providers will be to demonstrate compliance to the agreed upon QoS. We present a basic mechanism to continuously measure QoS in a federated Cloud environment so that resources can be provisioned or de-provisioned dynamically to meet Service Level Agreements. We have validated our mechanism by constructing prototypes and the results demonstrate that it is possible to continuously measure QoS at the minute granularity and for various service configurations prevalent in the industry.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Measurement, Performance, Design, Standardization

Keywords

Quality-of-Service, Service Level Agreements, Cloud Federation

1. INTRODUCTION

A major obstacle in Cloud computing is performance unpredictability because providers are unable to foresee temporal variations in service demands and the geographical distribution of their consumers [1]. Furthermore, no single Cloud provider is able to establish infrastructure large enough to support the perception of unlimited computing resources of the Cloud computing paradigm. For example, Amazon EC2 customer has a limit of 20 Reserved Instances per Availability Zone that they can purchase each month [2].These limitations will necessitate Cloud providers to engage in agreements with other Cloud providers to complement their own capacity. Cloud federation facilitates dynamic expansion and contraction of application services across multiple Clouds to achieve QoS targets under variable workload and computing resources. Due to this dynamic nature of the Cloud federation, continuous monitoring on QoS attributes is necessary to enforce Service Level Agreements (SLAs). We propose a basic mechanism for representing and measuring QoS of services across Cloud federations, accounting for various configurations such as single-point-of-failures, redundant services and planned down-times.

2. PROBLEM DESCRIPTION

The typical enterprise environment consists of thousands of systems per customer supporting around hundred services per client [3]. Each service, in turn, is supported by a diverse collection of systems consisting of web-servers, VMs, databases, application servers, storage, networking etc. SLAs are not only defined for each service but also for each of its sub-systems and their components. Service providers are contractually obligated to demonstrate compliance at each level of the service tree by an audit trail of calculation chain. This task of measuring and demonstrating SLA compliance becomes even more complex in a Cloud federation environment where service resources may be dynamically provisioned and de-provisioned within and across Cloud boundaries to handle sudden variations in service demands [1].

Existing techniques for measuring QoS usually employ centralized approaches to overall system monitoring and management. These centralized techniques are not an effective solution in a federation environment where live migration of virtual machines cross Cloud boundaries and challenges of auto-scaling and elasticity arise from unpredictable service demands [4].

To maximize cost-effectiveness and efficiency of composite systems it also becomes critical to allocate the optimal software and hardware configurations to ensure that QoS targets of services are achieved. This task of mapping services to resources becomes even more challenging in a Cloud federation model where expansion and resizing of provisioning capabilities are based on unforeseen spikes in workload demands in separate domains [4].

Enterprises with global operations will face difficulty in meeting QoS expectations for their entire user base because no single Cloud infrastructure provider has their data centers at all possible locations throughout the world. For example, Amazon has data centers in the US (e.g., one in the East Coast and another in the West Coast) and Europe. However, currently they expect their Cloud customers (i.e., SaaS providers) to express a preference about the location where they want their application services to be hosted [4]. As a result, Cloud providers would logically construct

federated Cloud infrastructure by mixing private and public Clouds.

To meet aforementioned requirements, services need to be more SLA-aware to clearly identify the boundaries of SLA violations and responsibilities. Since the QoS attributes change constantly in a dynamic environment, measurement and monitoring of system performance is required for dynamic allocation of resources in the changing environment of Cloud federation.

3. VISION

One of the biggest premises of Cloud computing is elastic scaling, which gives the users the perception of unlimited computing resources over the Internet. Cloud federation allows individual Cloud providers to engage in an agreement with other Cloud providers to enable elastic service composition which crosses Cloud boundaries [5]. Development of a basic mechanism to measure QoS is critical to complying with SLAs in a Cloud federation model where resource availability is uncertain. The most common QoS attributes which are part of SLA contracts are availability, response time and throughput of services. In a federated Cloud environment these metrics are constantly changing and they need to be monitored continuously to demonstrate compliance with the negotiated contracts.

We describe a basic mechanism for representing and measuring QoS of composed services across Cloud federations. We illustrate how this mechanism (Figures 1, 2, and 3) can measure 'up-to-the-minute' QoS for both individual services and composite services under various service configurations, such as single point of failure, redundant services (multiple servicer replicas) and services with planned downtime. Finally, we provide an algorithm (Figure 4) which uses this mechanism to optimize dynamic resource allocation within various service configurations.

4. REPRESENTATION

Figure 1 represents the basic mechanism for representing and measuring QoS of composed services across federations.

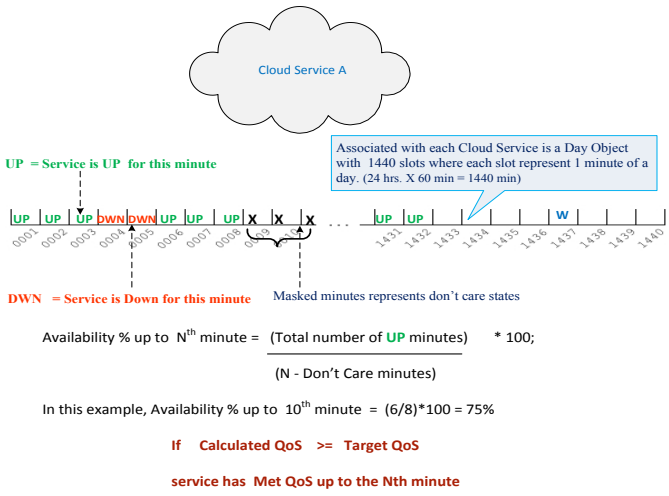


Figure 1. Basic Representation and Mechanism.

Associated with each service, is a Day object represented by an array consisting of 1440-minute objects representing each minute of a day (24 hours x 60 min = 1440). By default, each minute will be marked as 'UP' (or 1) minute. When a service disruption occurs during a particular minute the minute will be marked as a 'DWN' (or 0) minute. Additionally, a mask will be applied (indicated as X in the figure) to represent those minutes during which the user does not care whether the service is up or down. Figures 1 and 2 demonstrate the calculation of availability metrics up to the 10th minute. As long as the measured QoS is greater than or equal to the target QoS, the service would have met its expected target up to that minute. QoS of the entire Service tree and each of its components can be measured by recursively traversing the tree in a depth-first manner as explained in section 5. Also note how we assign a weight (W) vector to each minute to represent the component's relative contribution to the QoS of the composite service (parent) relative to other services for that minute. In Section 4.2 we explain how this weight vector mechanism can be used to calculate different QoS metrics under different service configurations such as composite service, single-points-of failures and services with redundancy built in.

4.1 Measuring Composite Cloud Services QoS

Figure 2 illustrates how the mechanism can be used to measure QoS attributes for composite services. In a federated environment, a composite service, for example an online publishing service, may be composed of other services such as editing, advertising, searching and printing services.

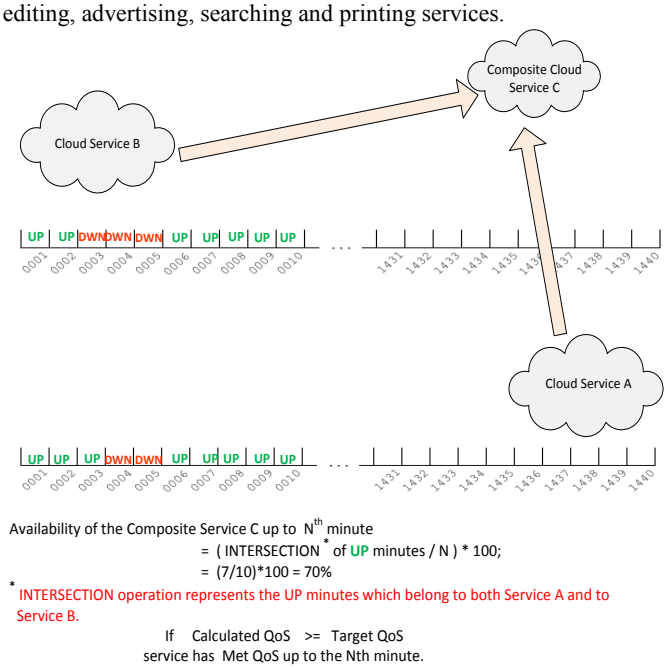


Figure 2. Continuous measurement of composite services.

For the composite service to be available, all the constituent services need to be available at the same time. This mechanism uses the set operation INTERSECTION (as depicted in the figure) to determine those minutes for which the composite service is available. The array of 1440 minutes is stacked on top of each other as the services are composed in a bottoms-up approach. For services which are redundant (e.g. two printing services) the set operation UNION is used.

4.2 Measuring Service Configurations QoS

In a service tree, when a child node contains a single-point-of-failure (e.g. the load balancer in clustered database system) the

parent service will incur an outage for every minute this single point of failure is down. However, if there is some redundancy built in (e.g. the database servers in the cluster environment), an outage of one or two servers may not impact the service. In essence, each child node is weighted differently with regard to how it impacts its parent node. The QoS impact calculations, therefore, need to differentiate between single-point-of-failures and those with redundancy built in. Additionally, a service may also be defined as 'impacted' only when n out of m servers providing the service are impacted. For example, when 3 out of the 5 database server goes down, then the system may incur serious latency issues, but when 2 out of those same 5 servers are down, the system may run without any problem.

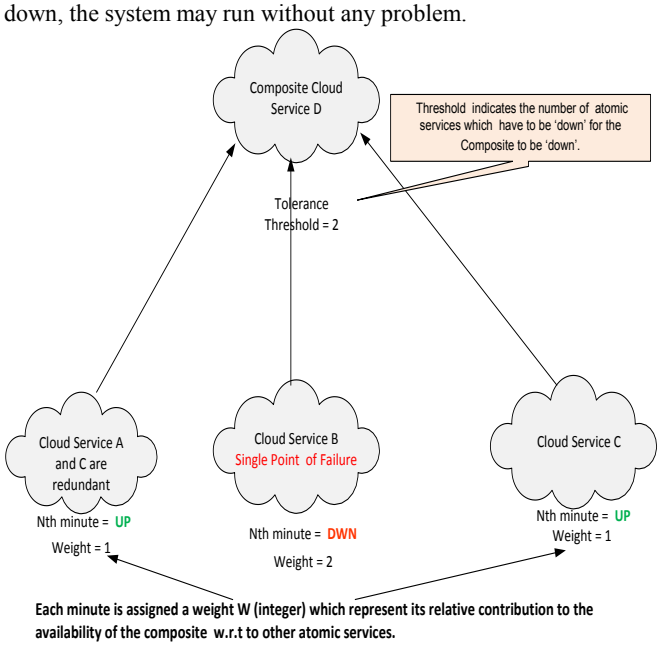


Figure 3. Use of Weight Vectors.

To account for these various service configurations, each node is given a 'weight' which indicates its criticality to the service it provides to its parent relative to other children. The parent is given a threshold tolerance level. This threshold tolerance indicates the level at which the parent service will incur an outage. The parent service will incur an outage when the sum of the weights of the children (incurring an outage) is greater or equal to this threshold tolerance of the parent.

Single-point-of-failure: The weight of these components is assigned the same value as the tolerance threshold of its parent. In this figure, Cloud Service B is the single-point-of-failure and it will take the Cloud service D down when it goes down (weight = threshold tolerance of parent).

Redundant services: Cloud service A and C are redundant services, so both of them have to be down for its parents to be down. Hence we assign weight such that the sum of their weight equals the tolerance threshold of its parent. Similarly, n out of m server configurations can also be accounted using this weight vector mechanism.

Service Disruptions in a Cloud federation: When service disruptions occur in a federated environment, it is necessary to accurately identify the offending cloud partition so that the other providers contributing to the same service are not penalized unduly. Furthermore, these disruptions must also be classified according to cause of disruption. For example, when a disruption is caused by anomalies in a third-party application hosted by an infrastructure cloud provider, the provider of the cloud infrastructure should not bear its consequence. To accurately identify and classify these types of service disruptions we associate configurable business rules to each of the minute objects. This enables filtering out problem tickets at the minute granularity.

QoS vary by cloud-partitions, providers and clients: One key observation in a federation model is that the service providers have to serve much more diverse clients than traditional enterprise services usually do [9]. To optimize profit, providers will service multiple clients with differing QoS from the same service tree. Furthermore, QoS levels may change at each level of the service tree for the same client. For example, a database service may have to be contractually available 99% of the time, while each of the servers providing the database service may have to be available for 95% of the time, while the load balancer for the servers needs to be up 99% of the time. To represent and measure QoS accurately, these threshold QoS metrics are stored in the day object (Figure 1) for each client and computations are carried out based on the information available in the service disruption for each client at each node in the SLA service tree.

Duplicate tickets elimination in a Cloud federation: Service disruption tickets can be created at device level, application level or at the service level. When SLAs are defined broadly at all these levels, often a secondary problem tickets are created to reflect the scenario more accurately. For example, the fact that a web service is down and one of the 2 servers supporting the service is down needs two distinct tickets to reflect reality [8]. In a federated environment where composite services are constructed from distinct cloud partitions, a mechanism needs to be in place so that providers are not penalized twice for the same outage. To detect duplicate tickets for the same outages, we stack outage minute objects from the problem tickets to the day object tray at each node of the service tree. If there are multiple outage minute objects for the same minute slot, then a duplicate ticket has been detected and they are rejected and logged for further analysis and reporting.

4.3 SLA-Based Provisioning Algorithm

Figure 4 demonstrates an SLA based provisioning algorithm to allocate resources for various service configurations. For each service and its components, set an Upper Bound QoS and a Lower Bound QoS threshold both of which are greater than the agreed upon QoS for that specific Service.

The range between the Lower Bound QoS and agreed QoS reflects a "near-breach" situation and alerts the provider of an impending violation of QoS. It reflects a service quality which has not yet violated the contractual QoS but it is close to breaching it. When the service quality falls below this lower bound, it provides an opportunity to add more resources or migrate the service to another cloud before a complete violation of QoS occurs.

In practice, expected QoS targets can vary from one node to another within the same service-tree. Hence, accurate measurement of both the target QoS and the up-to-the-minute QoS are necessary for all nodes within the service tree. This mechanism of measuring QoS continuously can be used to locate those services which are 'near-breaches' and based on the

criticality of the system (single-point-of-failure, redundant etc.), additional resources can be provisioned dynamically to prevent SLA violations. As an example, for single-point-of-failure, as soon as the measured 'up-to-the-minute' QoS falls below the upper bound, live migration to another local resource or to another Cloud in the federation is initiated. For services with redundancy built-in, this may occur only when the measured QoS falls below the lower bound.

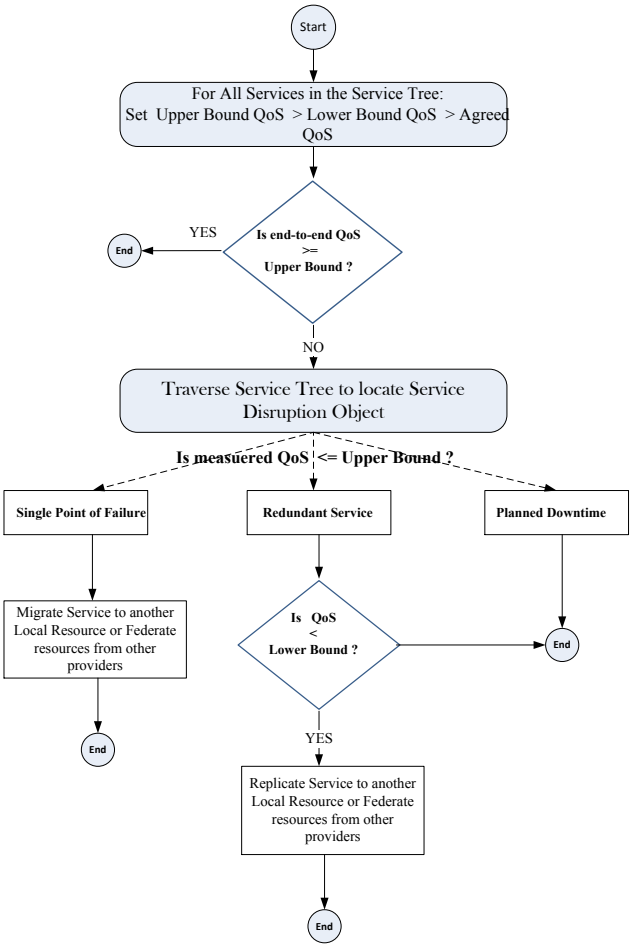


Figure 4. Algorithm for resource allocation, based on 'up-to-the-minute' QoS measurement of different service configurations.

Our assumption is that different cloud providers will either provide access to measure the service quality at each node or implement this technique at each node in their service tree. In order to speed-up processing, service quality at each node is pre-computed with the assumption that a service quality has met the contractual level unless a service disruption occurs. Only when a service disruption occurs, QoS computation is carried out using the representation described in Figure 1. In the software implementation, each minute is represented as an object which is capable of computing its own QoS value (e.g. availability percentage) based on several QoS attributes like down-time minutes, up-time minutes, caused-by field, scheduled/unscheduled outage etc.

'Up-to-the-minute' QoS measurement may not only prevent SLA violations but also optimize resource allocation by provisioning resources only when it is needed.

5. EXPERIMENTS

A prototype design was constructed and implemented to validate whether the mechanisms will hold up to various service configurations. The experiments were conducted on an HP server having the following configuration: 3.59 GHz with 4 GB of RAM running a standard Windows 2003 Server Standard Edition, Service Pack 2. The prototype environment was developed in C#, .NET 4.0 frameworks and the underlying database was SQL server 2008 in a load-balanced mode.

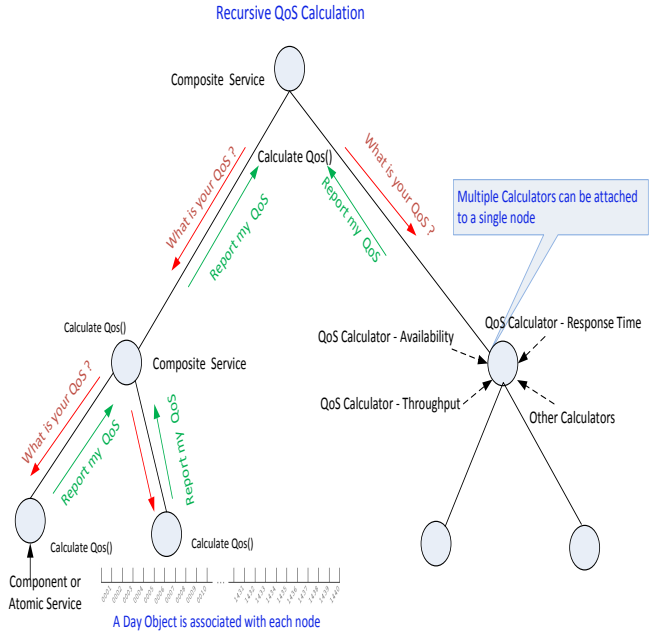


Figure 5. Recursive QoS calculation up/down the service tree.

The experiment was modeled after a real-world SLA service tree for a customer of Hewlett-Packard. The service tree (see Figure 5) consisted of 848 nodes where each node represented a server or a service. The typical service tree has the physical components (servers, network, and router) towards the leaf of the tree structure while the root represented the composite service. Each of the intermediate nodes represented a sub-system or a service which support the composite service at the root.

Associated with each node is the day object which is represented by an array of 1440 minutes objects to calculate up-to-the-minute QoS at every node. Starting at the root of the tree, the composite service calculates its QoS in a recursive manner based on the QoS of its children. Multiple Calculator Objects can be attached to a single node to compute different types of QoS at each node. The experiment was set up with single-point-of-failure and redundant services at different levels. While the computation logic was constructed at the application layer, the cumulative computation (up-to-the-minute) was done using stored procedure at the persistence layer to speed-up calculation.

The software program was run for each day for the month of June. The total number of tasks calculated was 1635 per day, since some nodes had multiple calculators associated with it. The task took 1.58 minutes to complete on one CPU. The results reveal that the software representation was able to make accurate calculation of the QoS for every minute for availability metrics at each node of the service tree. There were a total of 13 service

disruptions during the month of June and the software was able to compute the QoS in terms of availability percentage accurately.

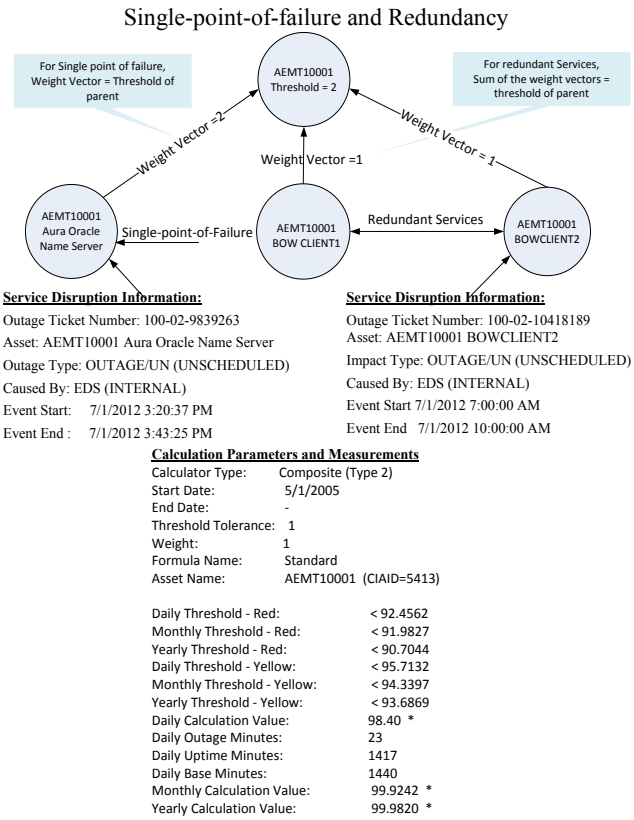


Figure 6. Weight Vectors Representing Service Configuration.

Figure 6 reveals actual calculations of SLA availability metrics for a customer of HP. There were two outages on the same day, one on a single-point-of-failure (left node) and one on a redundant server (right node). Since, two nodes on the right are redundant both of them have to incur an outage for the parent to be down. This is achieved by assigning weights to each of them such that their sum of the weight vectors equals the threshold tolerance of the parent. In this case, even though one of the redundant components was down for 3 hours, the parent was up because the sum of their weight of the down minutes is still less than threshold of parent. Also note, how the node on the left is a single-point-of-failure and its weight vector equals that of parents. Hence, the outage of 23 minutes it incurred was cascaded up to its parents.

5.1 Scaling-up in a Single-Datacenter

Experimental Setup: The setup is modeled after one of HP's clients in the travel and hospitality industry, whose assets are located in the Tulsa datacenters. The goal of the experiment was to find out if the measurement mechanism can scale-up when the number of nodes increases within an SLA service tree. The size of the service tree was steadily increased from 100 to 1200 and QoS computational time was recorded for each day, month-to-date, and year-to-date. The calculations were carried out for each minute for each day for a period of one month. The number of service disruptions during the same period was 19.

Results: The results reveal that the software representation of the mechanism scaled-up well when the size of the service tree was steadily increased. Figure 7 depicts a graph where the X axis represents the varying number of nodes in the service tree and the Y-axis represents the time it took to calculate up-time availability for a period of one month for the entire service tree. The result of the experiments also demonstrated that the computational time depended on the number of service disruptions for the period of measurement. This dependency can be explained by noting that the impact of the service disruption at any node needed to be cascaded up the service tree to calculate its impact on the end-to-end QoS.

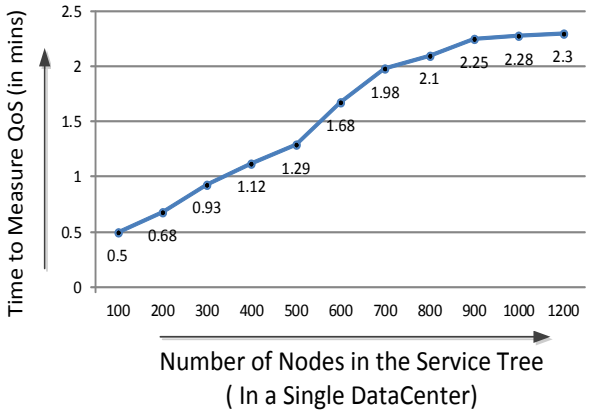


Figure 7. Measurements in a Single Datacenter.

5.2 Scaling-up in a Cloud Federation

Experimental Setup: Cloud partitions were simulated by carving out distinct computational environments within two geographically separate datacenters. The goal of the experiment is to find out if the measurement mechanism can also scale up in a cloud federation environment when the number of cloud partitions was increased. A service tree was modeled after one of HP's clients in the Transportation Industry whose assets are located in the Tulsa Datacenters. The size of the service tree was maintained fixed at 1200 nodes but they were equally distributed among each of the datacenter partitions. The number of partitions was varied from 1 to 8 and the QoS measurement software was run in each partition while sharing a single load-balanced database for storing results.

Results: The results of the simulation reveal that the software calculated the SLA performance metrics successfully in a federated environment. Figure 8 depicts a graph where the X axis represents the number of cloud partitions and the Y-axis represents the time it took to calculate the up-time availability for a period of one month for the entire service tree. As the number of cloud federations were increased the computational time increased linearly, which can be attributed to the fragmentation of service tree amongst different cloud partitions. The calculation results revealed that this measurement technique scaled up in a distributed environment as well. In the future, we intend to carry out this measurement technique in a real cloud federation where live migration of VM's and dynamic resource allocations can take place.

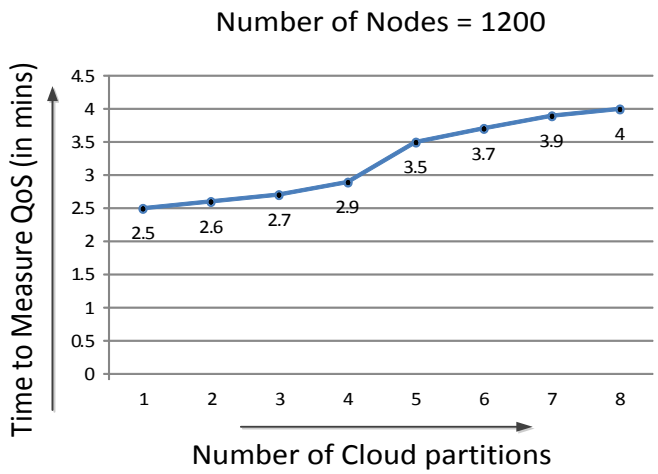


Figure 8. Measurements in a Cloud federation environment.

6. CONCLUSION AND FUTURE WORK

We presented a basic mechanism for representing and measuring QoS continuously for both individual service and composite services. The same mechanism can be used to represent and measure different QoS metrics such as availability, response time, throughput in a continuous manner in a Cloud federation. Development of this basic mechanism is critical to enabling elastic service composition in a Cloud federation model where resource availability is uncertain. We believe that the proposed measurement techniques will not only help in effective allocation of Cloud resources to satisfy QoS targets but also lower operational cost by enabling optimal resource pooling and surplus re-distribution in the highly dynamic federation model. Due to mounting concerns of trust and privacy, the task of measuring and demonstrating SLA compliance may be ultimately delegated to third-parties [7], who may adopt this clear and simple mechanism as a uniform standard of measurement.

In our future work, we shall examine how this continuous QoS measurement technique may prevent costly SLA violations by making services 'QoS-aware' so that autonomic resource management can occur in the changing environment of a Cloud federation.

7. REFERENCES

[1] Rajiv Ranjan, Rajkumar Buyya, Manish Parashar.2011. *Special section on autonomic Cloud computing: technologies, services, and applications*. Concurrency and Computation: Practice and Experience Special Issue: Special section on Autonomic cloud computing: technologies, services, and applications Volume 24, Issue 9, pp. 935–937, 25 June 2012

[2] *Amazon Elastic Compute Cloud (EC2),* http://aws.amazon.com/contact-us/reserved-instances-limit-request/

[3] Nicolas Bonvin, Thanasis G. Papaioannou and Karl Aberer. *Autonomic SLA-driven Provisioning for Cloud Applications.* 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. May 23-26, 2011, Newport Beach, CA, USA, pp. 1-5

[4] Rajkumar Buyya, Rajiv Ranjan, Rodrigo N. Calheiros. *InterCloud: Utility-Oriented federation of Cloud Computing Environments for Scaling of Application Services.* Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2010, Busan, South Korea, May 21-23, 2010), LNCS, Springer, Germany, pp. 14-20

[5] Stuart CLAYMAN, Alex GALIS, Clovis CHAPMAN, Giovanni TOFFETTI, 2010. *Monitoring Service Clouds in the Future Internet.* Towards the Future Internet G. Tselentis et al. (Eds.)IOS Press, 2010,pp.121-122

[6] R. Buyya, D. Abramson, and S. Venugopal. *The Grid Economy.* Special Issue on Grid Computing, Proceedings of the IEEE, M. Parashar and C. Lee (eds.), 93(3), IEEE Press, March 2005, pp. 698-714

[7] Pankesh Patel, Ajith Ranabahu, Amit Sheth. *Service Level Agreement in Cloud Computing.* Cloud Workshops at OOPSLA09, 2009, pp. 2-4

[8] Shoumen Bardhan, Steve Stevens. *Service Level Agreement Automation.* Hewlett-Packard TechCon'11, March 2011, pp 2

[9] {ychi,hjmoon,hakan,tatemura}@sv.nec-labs.com. *SLA-Tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing.* EDBT/ICDT '11 Proceedings of the 14th International Conference on Extending Database Technology Pages 129-140. 2011

Design of an Accounting and Metric-based Cloud-shifting and Cloud-seeding framework for Federated Clouds and Bare-metal Environments

Gregor von Laszewski[1]*, Hyungro Lee[1], Javier Diaz[1], Fugang Wang[1], Koji Tanaka[1], Shubhada Karavinkoppa[1], Geoffrey C. Fox[1], Tom Furlani[2]

[1]Indiana University
2719 East 10th Street
Bloomington, IN 47408. U.S.A.
laszewski@gmail.com

[2]Center for Computational Research
University at Buffalo
701 Ellicott St
Buffalo, New York 14203

ABSTRACT

We present the design of a dynamic provisioning system that is able to manage the resources of a federated cloud environment by focusing on their utilization. With our framework, it is not only possible to allocate resources at a particular time to a specific Infrastructure as a Service framework, but also to utilize them as part of a typical HPC environment controlled by batch queuing systems. Through this interplay between virtualized and non-virtualized resources, we provide a flexible resource management framework that can be adapted based on users' demands. The need for such a framework is motivated by real user data gathered during our operation of FutureGrid (FG). We observed that the usage of the different infrastructures vary over time changing from being over-utilized to underutilize and vice versa. Therefore, the proposed framework will be beneficial for users of environments such a FutureGrid where several infrastructures are supported with limited physical resources.

Categories and Subject Descriptors

D.4.8 [Performance]: Operational Analysis, Monitors, Measurements D.4.7 [Organization and Design]: Distributed systems

General Terms

Management, Measurement, Performance, Design, Economics.

Keywords

Cloud Metric, Dynamic Provisioning, RAIN, FutureGrid, Federated Clouds, Cloud seeding, Cloud shifting.

1. INTRODUCTION

Batch, Cloud and Grid computing build the pillars of today's modern scientific compute environments. Batch computing has traditionally supported high performance computing centers to better utilize their compute resources with the goal to satisfy the many concurrent users with sophisticated batch policies utilizing a number of well managed compute resources. Grid Computing and its predecessor meta-computing elevated this goal by not only introducing the utilization of multiple queues accessible to the users, but by establishing virtual organizations that share resources among the organizational users. This includes storage and compute resources and exposes the functionality that users need as services. Recently, it has been identified that these models are too restrictive, as many researchers and groups tend to develop and deploy their own software stacks on computational resources to build the specific environment required for their experiments. Cloud computing provides here a good solution as it introduces a level of abstraction that lets the advanced scientific community assemble their own images with their own software stacks and deploy them on large numbers of computational resources in clouds. Since a number of Infrastructure as a Service (IaaS) exist, our experience [1] tells us the importance of offering a variety of them to satisfy the various user community demands. In addition, it is important to support researchers that develop such frameworks further and may need more access to the compute and storage hardware resources than is provided by the current IaaS frameworks. For this reason, it is also important to provide users with the capabilities of staging their own software stack. This feature has also been introduced by other test-beds. This includes OpenCirrus [2], EmuLab [3], Grid5000 [4] and FutureGrid [5]. Within FutureGrid we developed a sophisticated set of services that simplify the instantiation of images that can be deployed on virtualized and non-virtualized resources contrasting our efforts.

The work described here significantly enhances the services developed and described in our publications about FutureGrid focusing on dynamic provisioning supported by image management, generation, and deployment [1] [6].

In this paper, we enhance our services in the following aspects:

a) Implementation of a uniform cloud metric framework for Eucalyptus 3 and OpenStack Essex.

b) Design of a flexible framework that allows resource re-allocation between various IaaS frameworks, as well as bare-metal.

c) Design of a meta-scheduler that re-allocates resources based on metric data gathered from the usage of different frameworks.

d) Targeted prototype development and deployment for FutureGrid.

The paper is organized as follows. In Section 2, we introduce the current state of Cloud Metrics as used in various IaaS frameworks. In Section 3 we give a short overview of FutureGrid. In Section 4 we present our elementary requirements that are fulfilled in our design, presented in Section 5. In Section 6 we outline the current status of our efforts and conclude our paper in Section 7.

2. ACCOUNTING SYSTEMS

Before we can present our design it is important to review existing accounting systems, as they will become an integral part of our design and solution. This not only covers accounting systems for clouds, but also for HPC and Grid computing as motivated by user needs and the ability of FutureGrid to provide a testbed for clouds, HPC, and Grid computing as discussed in more detail in Section 3.

Accounting systems have been put into production since the early days of computing stemming back to the mainframe, which introduced batch processing, but also virtualization. The purpose of such an accounting system is manifold, but one of its main purposes is to define a policy that allows the shared usage of the resources:

- *Enable tracking of resource usage* so that an accurate picture of current and past utilization of the resources can be determined and become an input to determining a proper resource policy.
- *Enable tracking of jobs and service usage by user and group* as they typically build the common unit of measurement in addition to the wall clock time as part of a resource allocation policy.
- *Enable a metric for economic charge* so that it can be integrated into resource policies as one input for scheduling jobs within the system.
- *Enable a resource allocation policy* so that multiple users can use the shared resource. The policy allows users to get typically a quota and establishes a priority order in which users can utilize the shared resource. Typically a number of metrics are placed into a model that determines the priority and order in which users and their jobs utilize the resource.
- *Enable the automation of the resource scheduling* task to a systems service instead of being conducted by the administrator.

One of the essential ingredients for such an accounting system are the measurements and metrics that are used as input to the scheduling model and is part of the active computer science research since 1960 with the advent of the first mainframes.

2.1 High Performance Computing

As High Performance Computing (HPC) systems have always been shared resources, batch systems usually include an accounting system. Typically metrics that are part of scheduling policies include number of jobs run by a user/group at a time, overall time used by a user/group on the HPC system, wait time for jobs to get started, size of the jobs, scale of the jobs, and more. Many batch systems are today available and include popular choices such as Moab which originated from Maui, SLURM [7], Univa Grid Engine [8] which originated from CODINE [9], PBS [10], and others.. Recently many of these vendors have made access to manipulation of the scheduling policies and the resource inventory, managed by the schedulers, much easier by adding Graphical user interfaces to them [10-12]. Many of them have also added services that provide cloud-bursting capabilities by submitting jobs for example to private or public clouds such as AWS.

One of the more popular accounting systems with the community is Gold [13]. Gold introduces an economical charge model similar to that of a bank. Transactions such as deposits, charges, transfers, and refunds allow easy integration with scheduling tools. One of the strength of Gold was its free availability and the possibility to integrate it with Grid resources. Unfortunately, the current maintainers of Gold have decided to discontinue its development and instead provide an alternative as a paid solution. It has to be seen if the community will continue picking up Gold or if they switch to the new system.

An interesting projects that has recently been initiated is the XDMod project [14] that is funded by NSF XD and is integrated into the XSEDE project. One of the tasks of this project includes the development of a sophisticated framework for analyzing account and usage data within XSEDE. However, we assume this project will develop an open source version that can be adapted for other purposes. This component contains an unprecedented richness of features to view, and creates reports based on user roles and access rights. It also allows the export of the data through Web services.

2.2 Grids

Accounting systems in Grids were initially independent from each other. Each member of a virtual organization had, by design, its own allocation and accounting policies. This is verifiable by the creation of the earliest virtual organization termed GUSTO [15], but also in more recent efforts such as the TeraGrid [16], XSEDE [17] and the OpenScience Grid [18]. Efforts were put in place later to establish a common resource usage unit to allow trading between resources, as for example in TeraGrid and XSEDE. The earliest metric to establish usage of Grid services outside of such frameworks in an independent fashion was initiated by von Laszewski et al. [19] for the usage of GridFTP and later on enhanced and integrated by the Globus project for other Grid services such as job utilization. Other systems such as Nimrod [20] provided a platform to the users in the Grid community that introduced economical metrics similar to Gold and allowed for the creation of trading and auction based systems. They have been followed up by a number of research activities [21] but such systems have not been part of larger deployments in the US.

2.3 Clouds

The de facto standard for clouds has been introduced by Amazon Web Services [22]. Since the initial offering, additional IaaS frameworks have become available to enable the creation of privately managed clouds. As part of these offering, we have additional components that address accounting and usage metrics. We find particularly relevant the work conducted by Amazon [23], Eucalyptus [24], Nimbus [25], OpenStack [26], and OpenNebula [27]. Other ongoing community activities also contribute in the accounting and metric area, most notably by integrating GreenIT [28, 29]. In addition, some of these cloud platforms can be enhanced by external *monitoring* tools like Nagios [30] and Ganglia [31].

For IaaS frameworks we make the following observations.

Amazon CloudWatch [23] provides real-time monitoring of resource utilization such as CPU, disk and network. It also enables users to collect metrics about AWS resources, as well as

publish custom metrics directly to Amazon CloudWatch. *Eucalyptus* enables, since version 3.0, usage reporting as part of its resource management [24, 32]. However, it does not provide a sophisticated accounting system that allows users and administrators to observe details about particular VM instances. To enable this third party tools such as Nagios, Ganglia and log file analyzers [24] have to be used. An integrated sophisticated and convenient framework for accounting is not provided by default.

Nimbus claims to support per-client usage tracking and per-user storage quota in its image repository and in Cumulus (Nimbus storage system) as accounting features. The per-client usage tracking provides information of requested VM instances and historical usage data. The per-user storage quota enables restriction of file system usage. Nimbus also uses Torque resource manager for gathering accounting logs. For monitoring features, Nimbus utilizes Nagios and Cloud Aggregator, which is a utility to receive system resource information.

OpenNebula has a utility named OpenNebula Watch [27] as an accounting information module. It stores activities of VM instances and hosts (clusters) to show resource utilization or charging data based on the aggregated data. OpenNebula Watch requires database handler like sequel, sqlite3 or MySQL to store the accounting information. It checks the status of hosts so physical systems can be monitored, for example, CPU and memory except network.

OpenStack is currently under heavy development in regards to many of its more advanced components. An on-going effort for developing accounting systems of OpenStack exists which is named Efficient Metering or ceilometer. It aims to collect all events from OpenStack components for billing and monitoring purposes [33]. This service will measure general resource attributes such as CPU core, memory, disk and network as used by the nova components. Additional metrics might be added to provide customization. Efficient Metering is planned to be released in the next version of Openstack (Folsom) late in 2012. Besides this effort, other metric projects include several billing projects such as Dough [34] and third party OpenStack billing plugin [35].

Microsoft Azure has a software called System Center Monitoring Pack that enables the monitoring of Azure applications [36]. According to Microsoft, the monitoring pack provides features such monitoring da and performance data, with integration to Microsoft supported products such as Azure, .NET. The performance monitoring can also be enabled by using some tools like Powershell cmdlets for Windows Azure [37] and Azure Diagnostics Manager 2 from Cerebrata [38]. The monitoring data can be visualized using System Center Operation Manager Console.

Google Compute Engine is an IaaS product launched end of June, 2012 and still under development [39]. Google currently supports several options for networking and storage while managing virtual machines through the compute engine. Presently, there is no accounting APIs for Google Compute Engine, but there is a monitoring API for Google App Engine. It delivers a usage report for displaying resource utilization of instances in the administration console [40] and provides a runtime API [41] to retrieve measured data from the application instances such as CPU, memory, and status. We expect that similar functionality will become available for the Google Compute Engine as well.

3. FUTUREGRID: AS A TESTBED FOR FEDERATED CLOUD RESEARCH

FutureGrid [42] provides a set of distributed resources totaling more than 4300 compute cores. Resources include a variety of different platforms allowing users to access heterogeneous distributed computing, network, and storage resources. Services to conduct HPC, Grid, and Cloud projects including various IaaS and PaaS are offered. Interesting interoperability and scalability experiments that foster research in many areas, including federated clouds, becomes possible due to this variety of resources and services. Users can experiment with various IaaS frameworks at the same time, and also integrate Grid and HPC services that are of special interest to the scientific community. One important feature of FutureGrid is that its software services can make use of the physical resources through both virtualization technologies and dynamic provisioning on bare-metal. This feature is provided by our software called *Rain* [1, 6], which allows us to *rain* a software stack and even the OS onto a compute server/resource. Authorized users have access to this feature that is ideal for performance experiments. Via the help of *Rain,* we can now devise a design and implementation that can re-allocate compute servers into various clouds determined by user demand. We refer to this new functionality as *cloud shifting*.

4. REQUIREMENTS

In [1] we presented qualitative and quantitative evidence that users are experimenting with a variety of IaaS frameworks. To support this need, we have instantiated multiple clouds based on multiple IaaS frameworks on distributed compute clusters in FG. However, the association of compute servers to the various IaaS frameworks is currently conducted manually by the system administrators through best effort. As this can become a labor intensive process, readjustments based on utilization needs occur infrequently, or not at all as some clusters have been dedicated to a particular IaaS framework regardless of utilization. However, our operational experience shows that readjustments are desirable while observing the usage patterns of over 240 projects hosted on FutureGrid. One such use pattern arises from educational classes in the distributed computing area. We observe that classes cycle through topics to teach students about HPC, Grid, and Cloud computing. When teaching cloud computing they also introduce multiple cloud IaaS frameworks. Thus, the demand to access the resources one after another is a logical consequence based on the way such classes are taught. However, this leads to resource starvation as at times certain services offered are underutilized, while others are over utilized.

Additionally, we observe that some projects utilize the resources in a federated fashion either while focusing on federation within the same IaaS framework [43], but more interestingly to federate between IaaS frameworks while focusing on scientific workflows that utilize cycle scavenging [44] or select frameworks that are most suitable for a particular set of calculations as part of the workflow [45]. These projects do not take into account that it is possible to conduct cloud shifting instead of scavenging resulting in a simplification of the development and utilization aspect for application developers.

In a coordinated response to our observations, we derived the following requirements that shape the design of the services in support of cloud federation research:

- *Support for multiple IaaS*: This includes OpenStack, Nimbus, Eucalyptus, and OpenNebula. Furthermore, we

would like to integrate with AWS and potentially other clouds hosted outside of FG.

- *Support for bare-metal provisioning to the privately managed resources*: This will allow us to *rain* custom designed software stacks on OS on demand onto each of the servers we choose.
- *Support for dynamic adjustment of service assignments*: The services placed on a server are not fixed, but can change over time via *Rain* [1, 6].
- *Support for educational class patterns*: Compute classes often require a particular set of services that are accessed by many of its members concurrently leading to spikes in the demand for one service type.
- *Support for advance provisioning:* Sometimes users know in advance when they need a particular service motivating the need for the instantiation of services in advance. This is different from advance reservation of a service, as the service is still shared by the users after the provisioning has taken place. Such a service will help to reduce resource starvation.
- *Support for advance reservation:* Some experiments require the exclusive access to the services.
- *Support for automation:* Managing such an environment should be automatized as much as possible.
- *Support for inter-cloud federation experiments:* Ability to access multiple IaaS instances at the same time.
- *Support for diverse user communities*: Users, Administrators, Groups, and services are interested in using the framework. These groups require different access rights and use modalities.

We intend to implement this design gradually and verify it on FG. The resulting software and services will be made available in open source so others can utilize them as well.

Due to these requirements we must support four very important functions of our framework. These functions include:

Cloud-bursting enables access to additional resources in other clouds when the demand for computing capacity spikes. It outsources services in case of over-provisioning, or inter-cloud federation enabling to use compute or storage resources across various clouds.

Cloud-seeding enables the instantiation of new cloud frameworks within FutureGrid.

Cloud-shifting enables moving (or re-allocating) compute resources between the various clouds and HPC.

Resource Provisioning is a basic functionality to enable cloud-seeding and -shifting as it allows the dynamic provisioning of the OS and software stack on bare-metal.

5. DESIGN

Before we explain our architecture, we have to point out some features of the resource and service fabric that are an integral part of our design. We assume that the *Resource Fabric* consists of a resource pool that contains a number of compute services. Such services are provided either as a cluster or as part of a distributed network of workstations (NOW). The resources are grouped based on network connectivity proximity. This will allow the creation of regions within cloud IaaS environments to perform more efficiently among its servers. We assume a rich variety of services offered in the *Service Fabric*. This includes multiple IaaS, PaaS frameworks, and HPC environments. Instead of assuming that there can only be one cloud for a particular IaaS framework, we envision multiple independent clouds. This assumption potentially allows users to host their own privately managed clouds and also integrate them with public clouds. We have already deployed such an infrastructure as part of the FutureGrid, allowing users to access a variety of preconfigured clouds to conduct interoperability experiments among the same IaaS and also different IaaS frameworks, as well as the inclusion of dedicated HPC services.

Having access to such a comprehensive environment opens up a number of interesting design challenges. We observe that our operational mode is significantly enhanced in contrast to other academic clouds that typically only install a single IaaS framework on their resource [46, 47]. Thus, such environments cannot offer by themselves the comprehensive infrastructure needed to conduct many of the topics that arise in cloud federation.

One of the questions we need to answer is how we can best utilize such an environment that supports inter-cloud and bare-metal demands posed by the users as we have practically observed in FutureGrid and how we can integrate these requirements into a software architecture.

We have designed a software architecture to address the requirements presented earlier. We distinguish the user layer allowing administrators, but also users (and groups of users) to interact with the framework. In addition, we point out that Web services can interact with it to develop third party automated tools and services leveraging the capabilities. Access to the various functions is provided in a secure fashion. Due to the diverse user communities wishing to use the environment, our design supports a variety of access interfaces including command line, dashboard, web services, as well as libraries and APIs.

An important aspect is to be able to integrate existing and future information services to provide the data to guide dynamic and automatic resource provisioning, cloud–bursting, cloud-seeding, and cloud-shifting. Due to this reason, we allow in our design the integration of events posted by services such as Inca, Ganglia, and Nagios. Moreover, we obtain information from the running clouds and, when the provided information is not sufficient, we will be able to ingest our own information by analyzing log files or other information obtained when running a cloud. For clouds, we also host an instance archive that allows us to capture traces of data that can be associated with a particular virtual machine instance. A metric archive allows the registration of a self-contained service that analyses the data gathered while providing a data series according to the metric specified. Metrics can be combined and can result in new data series.

At the center of this design is a comprehensive *RAIN* service Layer. *Rain* is an acronym for *R*untime *A*daptable *IN*sertion service signifying services that on the one hand adapt to runtime conditions and on the other allow inserting or dynamically provisioning software environments and stacks. We use the terms *rain* and *raining* to refer to the process of instantiating services on the resource and service fabrics. In this analogy, we can *rain* onto a cluster services that correspond to an IaaS, a PaaS, or a HPC batch system. *Rain* can be applied to virtualized and non-virtualized machine images and software stacks. In addition, *Rain* can also be used to move resources between already instantiated environments, hence supporting cloud-shifting. The most elementary operation to enable cloud-seeding

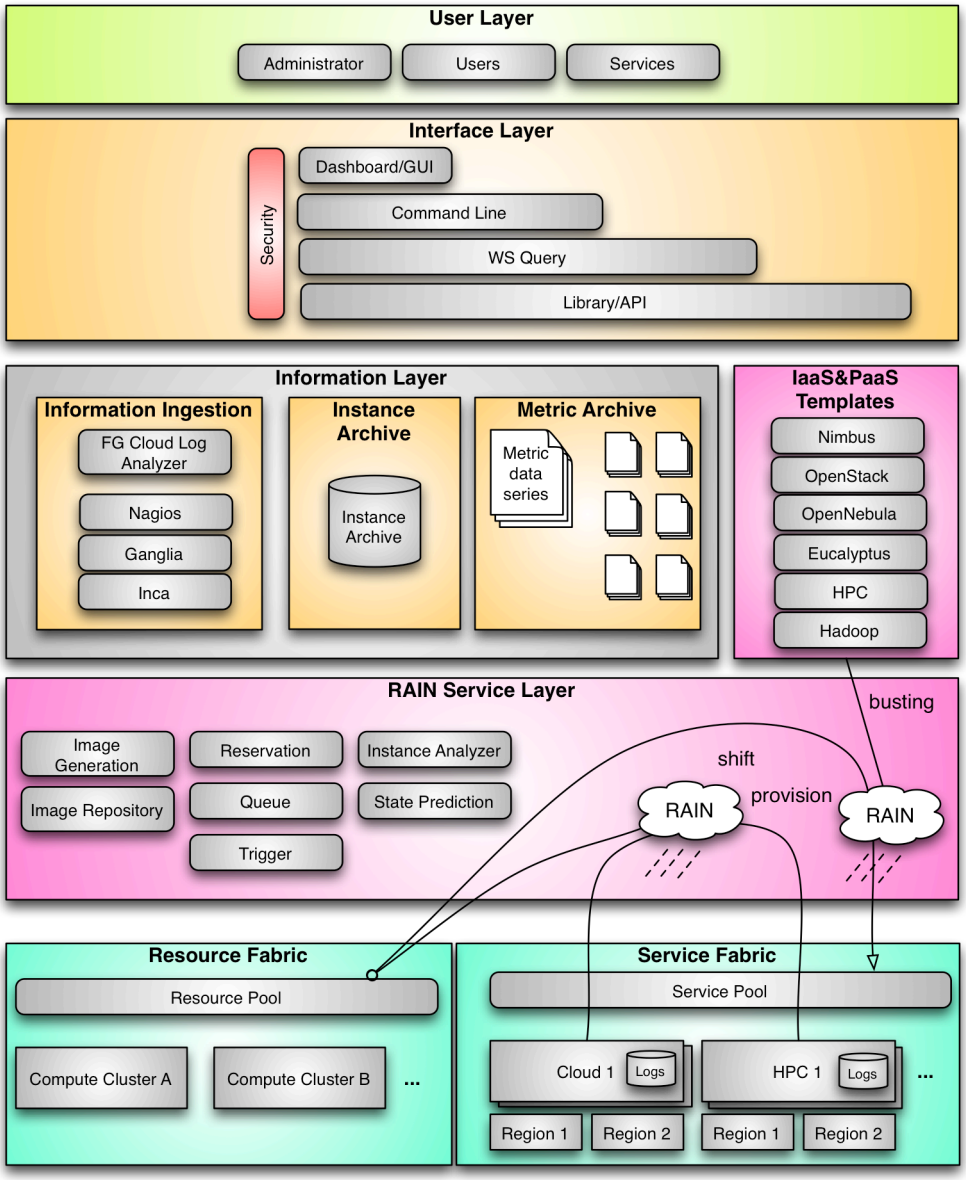


Figure 1: Design of the *rain*-based federated cloud management services.

and cloud-shifting is to provision the software and services onto the resources. We have devised this elementary operation and introduced in [6] and can now build upon it. In our past effort, we took on the problem of image management. In this work we focus on cloud-shifting, which is a logical extension in order to satisfy our users' needs.

Image Management. Rain allows us to dynamically provision images on IaaS and HPC resources. As users need quite a bit of sophistication to enable a cross platform independent image management, we have developed some tools that significantly simplify this problem. This is achieved by creating template images that are stored in a common image repository and adapted according to the environment or IaaS framework in which the image is to be deployed. Hence, users have the ability to setup experiment environments that provide similar functionality in different IaaS such as OpenStack, Eucalyptus, Nimbus, and HPC. Our image management services support the entire image lifecycle including generation, storage, reuse, and registration. Furthermore, we have started to provide extensions for image usage monitoring and quota management.

Cloud Shifting enables the re-allocation of compute resources within the various clouds and HPC. To enable cloud-shifting we have introduced a number of low-level tools and services that allow the re-allocation of resources from an IaaS or HPC service to another. A typical cloud-shifting request follows these steps:

1. Identify which resources should be moved (re-allocated) as part of the shift. This can be done by simply providing the names of the resources or by letting the service identify them according to service level agreements and specifiable requirements, such as using free nodes which have some specific characteristics.
2. De-register the resources from the service they are currently registered on. This involves identifying running jobs/VMs on the selected resource. If they exist, the service will wait a fixed amount of time for them to finish or it will terminate them. The behavior is selected when placing the request. Once a resource becomes available it will be placed into an *available resource pool*.

3. Pick resources from the available resource pool and *rain* the needed OS and other services onto each resource (if not already available).
4. Register the resources with the selected service and advertise their availability.
5. The resources are now integrated in the service and can be used by the users.

Cloud-Seeding allows us to deploy a new service such as cloud infrastructure, from scratch, in FutureGrid. Thus, cloud-seeding enhances and makes use of the previously described cloud-shifting. A cloud-seeding request includes:

1. Install the new service (cloud or HPC infrastructure) in the selected resources.
2. Set up the new service with some predefined configuration or following some given specifications. These specifications may include private IP range, ACL policies, and users' profiles.
3. Make use of cloud-shifting to add resources to the newly created service.
4. Announce and advertise the availability of the new service to the users and other services.

This is not a simple process, because it requires a great deal of planning and knowledge about the available infrastructure. Currently, we do this planning step by hand, but we intend to further automatize it as much as possible

Queue Service. We anticipate that users may have demands that cannot be immediately fulfilled by using a single request of the cloud shifting or cloud seeding services. Therefore, our design includes the introduction of a queuing service that can coordinate multiple requests. In this way, users can create a workflow that will subsequently call different services to create the desired environment. Such a queuing service could be hidden from the users and integrate with cloud-bursting services to integrate additional resources in case of overprovisioning. Once no additional resources are available requests are queued.

Reservation Service. Our design also includes the introduction of a reservation service to satisfy users with definite requests to be fulfilled at predefined times. This is the case for tutorials, classes, and regularly executed experiments.

State Prediction Service. This service will provide accounting and usage information, as well as, access to customized metrics while monitoring our different services to predict their anticipated usage. For cloud IaaS frameworks, our instance database and instance analyzer (that we developed) will collect valuable input of the resource and service fabrics.

Metrics. Elementary inputs to our prediction service are the metrics to guide our framework. These metrics are fed by elementary information in regards to job and virtual machine traces.

Traditional computing systems provide common resource metrics such as CPU, memory, storage, network bandwidth, and electricity utilization.

In case of VMs, we have to expand these metrics with VM specific information such as VM state, size, type, OS, memory, disk, CPU, kernel, Network IP, owner, and label. In addition, we are concerned with how much time it costs to create the VM, transfer it to a resource, instantiate and dynamically provision it, as well as bringing it in a state that allows access by the user. Furthermore, once the machine is shut down, we need to account

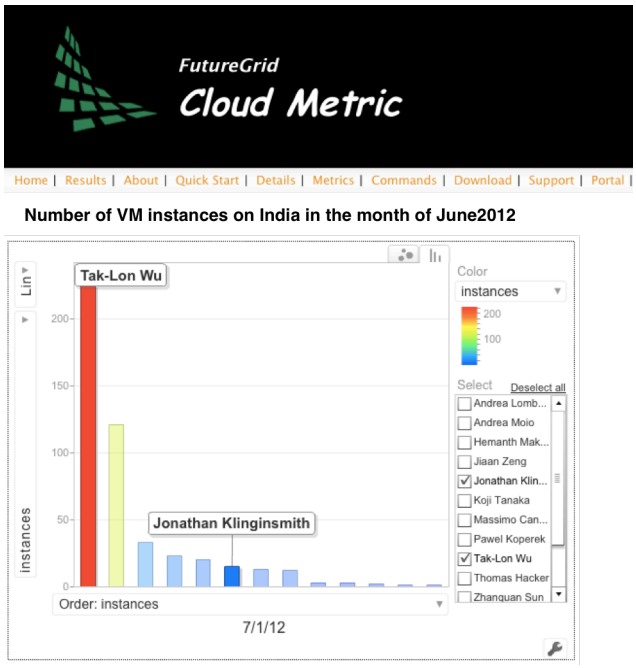


Figure 2: Screenshot of our Cloud Instance Analyzing

for the shutdown time and eventual cleanup or removal of the VM. Naturally we also need to keep track of which user, group or project instantiated the VM and if the image is a replication run in parallel on other resources in the fabric.

When dealing with services that are dependent on performance metrics, we also have to deal with periodicity of the events and filter out events not only based potentially on a yearly, monthly, weekly, daily, hourly, minute or per second basis, but to eliminate events that do not contribute significantly to the trace of a virtual machine image. We have practically devised such a service for Eucalyptus that reduced four million log events to about 10000 trace events for virtual machine images. This allows us to query needed information for our predictive services in milliseconds rather than hours of reanalyzing such log entries over and over again. Hence, our design is not only to retrieve standard information such as average, sum, minimum and maximum, as well as count of VM related events, but it can also input this data efficiently into a time series analysis and predictive service. In addition, we have integrated metrics for OpenStack and are in the process of expanding to Nimbus. Clearly, this framework is a sophisticated tool in support of federated heterogeneous and homogeneous clouds.

6. STATUS AND IMPLEMENTATION

As already pointed out, we have developed the basic infrastructure to support *rain* by enabling the image management services. These services are in detail documented in [1, 6, 42]. Recently we started the development of *rain* services that address the issue of cloud-shifting. We developed the ability to add and remove resources dynamically to and from Eucalyptus, OpenStack and HPC services. This allows us to easily move resources between OpenStack, Eucalyptus, and HPC services. We used this service to shift resources in support of a summer school held at the end of July 2012 where more than 100 participants were taught a variety of IaaS and PaaS frameworks. Within a short period of time we were able to adapt

our resource assignments to more closely serve the requirements of the projects executed at the time. As currently, some features in Nimbus are missing that are necessary to integrate with our framework, FutureGrid is also funding the Nimbus project to enhance their services so they will allow similar features as other IaaS frameworks already provide in order to support our image management framework. In parallel, we have significantly contributed towards the analysis of instance data for Eucalyptus and OpenStack clouds. Such data is instrumental for our planned predictive services. This effort includes the creation of a comprehensive database for instance traces that records important changes conducted as part of the VM instance runtime documented in our design section. A previous analysis effort that analyses log files in a repeated fashion was designed and implemented by von Laszewski, Wang, and Lee, replacing an effort that allows the ingestion and extraction of important log files from newly created log events [48]. As a result, we were able to significantly reduce the log entries, which led to a speedup of our analyzing capabilities from hours to milliseconds. In addition, we made the framework independent from web frameworks and chart display tools. A convenient command shell that can also be accessed as a command line tool was added to allow for interactive sampling and preparation of data. Web services, as well as a simple graphical interface to this data will be available (see Figure 2). At the same time the code was significantly reduced and modularized so that future maintenance and enhancements become easier. Examples for data currently presented in our Web interface are based on the utilization of several metrics. This includes total running hours of VM instances; total number of VM instances in a particular state and time interval; CPU core, memory and disk allocations; delay of launching and termination requests, the provisioning Interval, and geographical locations of VM instances. Metrics projecting a per user, per group, or per project view, metrics per cloud view, as well as metrics for the overall infrastructure and metrics related to the resource and service fabric are under development. Additional metrics such as traffic intensity for a particular time period [49, 50] are also useful in considering optimized utilization. Future activities will also include our strategy to use DevOps frameworks that we started from the beginning of the project and have also been independently been used by the FutureGrid community [51]. Clearly our framework can also be beneficial for integrative cloud environments such as CometCloud [52].

7. CONCLUSION

In this paper, we have presented a design of a federated cloud environment that is not focused singly on supporting just an IaaS framework. Our understanding of federation includes various IaaS frameworks on potentially heterogeneous compute resources. In addition, we are expanding our federated cloud environment to include and integrate traditional HPC services. This work is a significant enhancement to our earlier work on dynamic image generation and provisioning in clouds and bare-metal environments by addressing challenges arising in cloud seeding and cloud shifting. One of the other contributions of this paper is the creation of an accounting and metric framework that allows us to manage traces of virtual machine instances. This framework will be an essential component towards automating cloud-shifting and seeding as projected by our architectural design. We welcome additional collaborators to contribute to our efforts and to use FutureGrid.

8. ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812 and 1025159. We like to thank the members of FG for their help and support. We like to thank the Eucalyptus team for their willingness and excellent help in letting us use their commercial product.

9. REFERENCES

[1] von Laszewski, G., Diaz, J., Wang, F. and Fox, G. C. Comparison of Multiple Cloud Frameworks. In *Proceedings of the IEEE CLOUD 2012, 5th International Conference on Cloud Computing* (Honolulu, HI, USA, 24-29 June, 2012). Doi 10.1109/CLOUD.2012.104.

[2] Avetisyan, A. I., Campbell, R., Gupta, I., Heath, M. T., Ko, S. Y., Ganger, G. R., Kozuch, M. A., O'Hallaron, D., Kunze, M., Kwan, T. T., Lai, K., Lyons, M., Milojicic, D. S., Hing Yan, L., Yeng Chai, S., Ng Kwang, M., Luke, J. Y. and Han, N. Open Cirrus: A Global Cloud Computing Testbed. *Computer*, 43, 4 2010), 35-43. Doi 10.1109/mc.2010.111.

[3] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C. and Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Proceedings of the 5th Symposium on Operating Systems Design & Implementation* (December 2002, 2002).

[4] Grid5000 Home Page, http://www.grid5000.fr/.

[5] G. von Laszewski, G. C. Fox, Fugang Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Vöckler, R. J. Figueiredo, J. Fortes and K. Keahey. Design of the FutureGrid experiment management framework. In *Proceedings of the Gateway Computing Environments Workshop (GCE) at SC10* (New Orleans, LA, 14-14 Nov. 2010, 2010). IEEE. Doi 10.1109/GCE.2010.5676126

[6] Diaz, J., von Laszewski, G., Wang, F. and Fox, G. Abstract Image Management and Universal Image Registration for Cloud and HPC Infrastructures. In *Proceedings of the IEEE CLOUD 2012, 5th International Conference on Cloud Computing* (Honolulu, HI, 2012). IEEE. Doi 10.1109/cloud.2012.94.

[7] Yoo, A., Jette, M. and Grondona, M. SLURM: Simple Linux Utility for Resource Management. in Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol 2862, p. 44-60, Springer Berlin / Heidelberg, 2003.

[8] Univa Grid Engine, http://www.univa.com/products/grid-engine/.

[9] Genias CODINE: Computing in distributed networked environments (1995), http://www.genias.de/genias/english/codine.html.

[10] Altair PBS, http://www.pbsworks.com/.

[11] Moab, http://www.adaptivecomputing.com/products/hpc-products/.

[12] Bright-Computing Cluster Manager, http://www.brightcomputing.com/Bright-Cluster-Manager.php.

[13] Adaptive-Computing Gold Allocation Manager User Guide, http://www.adaptivecomputing.com/resources/docs/gold/.

[14] XDMoD XDMoD (XSEDE Metrics on Demand), https://xdmod.ccr.buffalo.edu/.

[15] Globus-Project Globus Ubiquitous Supercomputing Testbed Organization (GUSTO), http://www.startap.net/PUBLICATIONS/news-globus2.html.

[16] Hart, D. Measuring TeraGrid: Workload Characterization for an HPC Federation. *International Journal of High Performance Computing Applications* 4(Nov. 2011), 451-465. Doi 10.1177/1094342010394382.

[17] XSEDE: Extreme Science and Engineering Discovery Environment, https://http://www.xsede.org.

[18] OSG Open Science Grid, http://www.opensciencegrid.org.

[19] von Laszewski, G., DiCarlo, J. and Allcock, B. A Portal for Visualizing Grid Usage. *Concurrency and Computation: Practice and Experience*, 19, 12 (presented in GCE 2005 at SC'2005 2007), 1683-1692. Doi

[20] Abramson, D., Foster, I., Giddy, J., Lewis, A., Sosic, R., Sutherst, R. and White, N. The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing. In *Proceedings of the Proceedings of the 20th Australasian Computer Science Conference* (1997).

[21] Buyya, R., Yeo, C. S. and Venugopal, S. *Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities, in*. 2008.

[22] Amazon Amazon Web Services, http://aws.amazon.com/.

[23] Amazon Web Services Cloud Watch, http://docs.amazonwebservices.com/AmazonCloudWatch/latest/DeveloperGuide/CloudWatch_Introduction.html.

[24] Eucalyptus Eucalyptus Monitoring, http://open.eucalyptus.com/wiki/EucalyptusMonitoring_v1.6.

[25] Nimbus-Project Per Client Tracking, http://www.nimbusproject.org/docs/current/features.html.

[26] OpenStack Multi-Tenant Accounting, http://wiki.openstack.org/openstack-accounting?action=AttachFile&do=get&target=accounts.pdf.

[27] OpenNebula OpenNebula Watch - Accounting and Statistics 3.0, http://opennebula.org/documentation:archives:rel3.0:acctd_conf.

[28] Beloglazov, A., Buyya, R., Lee, Y. C. and Zomaya, A. A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems2010). Doi

[29] Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q. and Pentikousis, K. Energy-Efficient Cloud Computing2010). Doi

[30] Barth, W. *Nagios: System and Network Monitoring*. No Starch Press, San Francisco, CA, USA, 2006.

[31] Massie, M. L., Chun, B. N. and Culler, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30, 7 2004), 817 - 840. Doi 10.1016/j.parco.2004.04.001.

[32] Eucalyptus Eucalyptus 3.0.2 Administration guide, http://www.eucalyptus.com/docs/3.0/ag.pdf.

[33] OpenStack Blue print of Efficient Metering, http://wiki.openstack.org/EfficientMetering.

[34] Luo, Z. Dough, https://github.com/lzyeval/dough.

[35] OpenStack Billing Plugin for OpenStack, https://github.com/trystack/dash_billing.

[36] Microsoft Introduction to the Monitoring Pack for Windows Azure Applications

[37] Microsoft Windows Azure PowerShell Cmdlets, http://wappowershell.codeplex.com/.

[38] Red-Gate-Software Cerebrata, http://www.cerebrata.com/.

[39] Google Google Compute Engine, http://en.wikipedia.org/wiki/Google_Compute_Engine.

[40] Google Monitoring Resource Usage of Google App Engine, https://developers.google.com/appengine/docs/python/backends/overview#Monitoring_Resource_Usage.

[41] Google Runtime API for Google App Engine, https://developers.google.com/appengine/docs/python/backends/runtimeapi.

[42] von Laszewski, G., Fox, G. C., Wang, F., Younge, A. J., Kulshrestha, A., Pike, G. G., Smith, W., Voeckler, J., Figueiredo, R. J., Fortes, J., Keahey, K. and Deelman, E. Design of the FutureGrid experiment management framework. In *Proceedings of the Gateway Computing Environments Workshop (GCE), 2010 in conjunction with SC10* (New Orleans, LA, 14-14 Nov. 2010, 2010). IEEE. Doi 10.1109/GCE.2010.5676126.

[43] Keahey, K., Tsugawa, M., Matsunaga, A. and Fortes, J. Sky Computing. *Internet Computing, IEEE*, 132009), 43-51. Doi 10.1109/MIC.2009.94.

[44] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C. and Katz, D. S. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13, 3 2005), 219-237. Doi 10.1.1.117.132.

[45] Thilina Gunarathne, Judy Qiu and Geoffrey Fox. Iterative MapReduce for Azure Cloud In *Proceedings of the CCA11 Cloud Computing and Its Applications* (Chicago, IL, April 12-13, 2011).

[46] Cornell Cornell University Red Cloud, http://www.cac.cornell.edu/redcloud/.

[47] Clemson Clemson University One Cloud, https://sites.google.com/site/cuonecloud/.

[48] Laszewski, G. v., Lee, H. and Wang, F. Eucalyptus Metric Framework (Source Code), https://github.com/futuregrid/futuregrid-cloud-metrics.

[49] Calheiros, R. N., Ranjan, R. and Buyya, R. Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments. In *Proceedings of the International Conference on Parallel Processing* (Washngton, DC, 2011). IEEE Computer Society. Doi 10.1109/ICPP.2011.17.

[50] Maguluri, S. T., Srikant, R. and Ying, L. *Stochastic Models of Load Balancing and Scheduling in Cloud Computing Clusters*. 2012.

[51] Klinginsmith, J., Mahoui, M. and Wu, Y. M. Towards Reproducible eScience in the Cloud. In *Proceedings of the Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (Nov. 29 2011-Dec. 1 2011, 2011). Doi 10.1109/CloudCom.2011.89.

[52] Kim, H. and Parashar, M. CometCloud: An Autonomic Cloud Engine. in Cloud Computing, vol p. 275-297, John Wiley & Sons, Inc., 2011.

Infrastructure Outsourcing in Multi-Cloud Environment

Kate Keahey
Argonne National Laboratory
keahey@mcs.anl.gov

Patrick Armstrong
University of Chicago
oldpatricka@uchicago.edu

John Bresnahan
Argonne National Laboratory
bresnaha@mcs.anl.gov

David LaBissoniere
University of Chicago
labisso@uchicago.edu

Pierre Riteau
University of Chicago
priteau@uchicago.edu

ABSTRACT

Infrastructure clouds created ideal conditions for users to outsource their infrastructure needs by offering on-demand, short-term access, pay-as-you-go business model, the use of virtualization technologies which provide a safe and cost-effective way for users to manage and customize their environments, and sheer convenience, as users and institutions no longer have to have specialized IT departments and can focus on their core mission instead. These key innovations however also bring challenges which include high levels of failure; lack of interoperability between cloud providers, which puts significant lock-in pressure on the user, and lack of tools that allow users to leverage the on-demand growing and shrinking of infrastructure. All these factors prevent users from capitalizing on the infrastructure cloud opportunity. In this paper we propose a multi-cloud auto-scaling service that enables the user to leverage "computational power on tap" provided by infrastructure clouds, i.e., allows the user to easily deploy resources across multiple private, community, and commercial clouds; provides high availability in that it allows users to replace failed resources; and scales to demand. The policies governing scaling are customizable based on system and application-specific indicators. We will describe the service architecture and implementation and discuss results obtained in the sustained deployment and management of thousands of virtual machines on EC2.

General Terms

Management, Design, Experimentation.

Keywords

Cloud computing, Infrastructure-as-a-Service, Platform-as-a-Service, Nimbus.

1. INTRODUCTION

Outsourcing and sharing resources has many potential benefits to scientific projects. First, it provides access to more sophisticated resources -- in terms of size, cutting-edge technology or architectural diversity -- that is often beyond the means of a single institution to acquire. The resource can also be used with greater flexibility: e.g., a small "slice" of resource over long time or a larger slice occasionally. Further, it creates potential for access to economies of scale via consolidation and thus provides a better amortization of the original investment. And, last but not least, it eliminates of the overhead of system acquisition and operation for an institution via outsourcing computing. This is valuable as it allows scientific institutions to focus on delivering results in the form of scientific breakthroughs that are its mission in the first place. All those reasons are powerful motivators for outsourcing where a suitable outsourcing paradigm can be found.

Workshop on Cloud Computing Federation, September 21st, 2012, San Jose, CA

Infrastructure-as-a-Service (IaaS) clouds [1] (also called infrastructure clouds) created ideal conditions for users to outsource their infrastructure needs. A typical infrastructure cloud offers (1) on-demand, short-term access, which allows users to flexibly manage peaks in demand, (2) pay-as-you-go model, which helps save costs for bursty usage patterns (i.e., helps manage "valleys" in demand), (3) access via virtualization technologies which provides a safe and cost-effective way for users to manage and customize their own environments, and (4) sheer convenience, as users and institutions no longer have to have specialized IT departments and can focus on their core mission instead. The flexibility of the approach allows users to also outsource as much or as little of their infrastructure procurement as their needs justify: they can keep a resident amount of infrastructure in-house while outsourcing only at times of increased demand and they can outsource to a variety of providers choosing the best service levels for the price the market has to offer.

However, as well as all these advantages the cloud environment also has its challenges. The clouds existing today have levels of failure that are higher than traditional in-house machines; this is due not only to service levels of specific providers, but also to the fact that infrastructure cloud resources are typically accessed over the Internet with high potential for network failures or delays. Interoperability between providers is in its infancy, making efficient markets infeasible and putting significant lock-in pressure on the user (i.e., the barrier for moving from provider to provider is significant and the effort to scale it is borne entirely by the user). And last but not least, currently the methods of leveraging the on-demand growing and shrinking of infrastructure are crude to say the least: users typically are reduced to doing it manually or setting up primitive infrastructures to manage part of the solution space for them. This prevents them from capitalizing on the infrastructure cloud opportunity.

In this paper, we describe infrastructure which automates outsourcing computation to the cloud. We propose a multi-cloud auto-scaling service that provides an easy way for the user to leverage "computational power on tap", i.e., allows the user to easily deploy resources across multiple private, community, and commercial clouds. Our emphasis is on enabling users to build highly scalable services and also provide high availability, to allow users to replace failed resources ensuring continuous presence on the network. Further, the system carries support for adapting policies governing scaling based on system and application-specific indicators. We will describe the service architecture and implementation and discuss results obtained in

the sustained deployment and management of thousands of virtual machines on Amazon Web Services' EC2 service [2].

This paper is structured as follows. Section 3 describes the model our services implement. Section 4 and 2 describe respectively the architecture and the design principles that led to the development of this architecture. Section 5 describes challenges in scalability and their resolution when implementing scaling of thousands of VM instances on Amazon's EC2.

2. DESIGN PRINCIPLES

Below we summarize the principles behind our design:

Any Scale. To ensure "computational power on tap" it is necessary to allow users scale the resources over which their computation is deployed, up and down, easily and automatically – much as electrical devices have the ability to draw as much or as little power from the grid as they need. We should provide the ability to auto-scale via dynamically provisioning resources in the cloud in reaction to system-specific or application-specific sensors, which the user can pick from a library of ready-made, pre-defined sensors, but also give the user the ability to define and add his or her own sensors that can eventually be published and shared with other users. The sensors should include a large variety of events -- including operator-driven/manual events. This implies information presentation consistent with manual control: the ability to "zoom in" on the state and composition of any collection of resources, as well as the ability to use simple visual cues such as e.g., up and down arrow keys to regulate resource size.

High Availability (HA). Infrastructure clouds today are often unreliable [3]. This is due not only to service levels of specific providers, but also to the fact that infrastructure cloud resources are typically accessed over the Internet with high potential for network failures or delays – and as such may only improve in special cases in the foreseeable future. For this reason, it is essential that an infrastructure aiming to deliver "uninterrupted power supply" address this problem. We develop systems with the assumption that any VM can die and be replaced quickly to preserve the system from a prolonged downtime or service level deterioration. For this reason, we design the system to achieve minimum time to repair (TTR).

MultiCloud. Any system that provisions resources from only one provider exposes itself to the same kind of failure that threatens electricity consumer taking power from just one source. From the perspective of achieving "uninterrupted power supply" working with multiple providers isolates the consumer from technical and business failures any one provider might be experiencing and thus prevents vendor lock-in. It also provides the underpinnings for the creation of markets and thus conveying the best price to the consumer via competition. For this reason our infrastructure should allow users to integrate resources from multiple infrastructure clouds: from private clouds to community and commercial clouds as a "continuum" of available infrastructure resources.

Your Policies, Our Enactment. Different applications and services want to use cloud resources differently. Our focus is on providing infrastructure, i.e., solid and robust enactment that can carry out user policies taking into accont resource availability and status. Providing policies should be as broad as possible, e.g. it should be possible to extend the system by building customized policy plugins. The system will provide the mechanism to collect all the sensor inputs that could be relevant for the user to develop relevant policies and also allow the user to add custom sensors.

3. MODEL

We define a worker VM as a single, ephemeral and replacable compute resource that can be provisioned on-demand and capable of carrying out computation of a specific kind. Worker VMs are ephemeral by which we mean that they may become unavailable at any time, due to resource failure, temporary or persistent network failure, or other issues. A worker unit also needs to be replaceable, i.e. it should be possible to replace it with a worker unit of the same kind such that the new unit can automatically rejoin the computation carried out by the worker units. The worker units are independent of each other.

We define a domain as a pool of homogeneous worker VMs that are contextualized (in terms of both security and configuration) to work with domain-specific entities and whose size is governed by domain-specific policies. There could be potentially many distinct domains composed of VMs of the same type, belonging to different clients or serving different purposes for those clients, just like there are many potential instantiations of a service.

The policies governing the size of the domain may take into account multiple factors including system-specific metrics, such as number of individual worker units or load on individual worker units, application-specific metrics, such as the size and makeup of the workload queue or the number of network connections open to a specific resource, or a variety of other events including e.g., console events or an explicit request by a domain stakeholder (such as a scheduler for example). At any given time, due to a variety of conditions (such as changes in policy, worker node failures or inaccessibility), a domain has an intended size and an actual size – the purpose of a domain manager is to ensure that the intended size is equal to the actual size.

3.1 Using Domains

Our model is based on the assumption that the work of a service can be performed on multiple VMs independent of each other, that can be scaled up or down according to demand. This means that an application or service leveraging this model has to be capable of absorbing new processing capability, i.e., an application process that is newly strated can be automatically integrated into the application. The concept of high availability is additionally reliant on the fact that any process may die without affecting the application as a whole. This means that the work carried out on the VMs is such that can be easily interrupted and easily taken over by a replacement unit should a resource failure occur.

The most common way to leverage the concept of a domain is via applications with a well structured, ordered, workload. Such workload can be represented by an AMQP message queue, where the messages represent work units, a scheduler queue, where jobs represent work units, a workflow of tasks to be executed on a domain, or a list of data transfers to be serviced in a data transfer service. Workload is a logical concept and does not necessarily imply a durable queue. In practice, a domain could be used to support e.g., web servers that work with DNS round-robin. Howeer, in order to provide high availability we have to rely on a durable queue, i.e., with the property that a work unit persists on the queue until its receipt has been acknowledged. A workload can represent a "push queue", which directs units of work to specific worker units in a domain (typically implemented by e.g., batch schedulers or workflow systems) or a "pull queue" which relies on the worker units to claim the work units themselves (implemented by e.g. systems such as BOINC [4] or Condor [5]).

In our model, a service is realized by conveying the workload from the service clients. On the interface side, the

clients add their work units to the workload queue and get notified of the acceptance of the request and eventually a result. On the execution side, the workload is distributed to worker units of a domain hosting the execution of the service (either by the workers accessing the workload queue themselves in a pull queue, or by the work units being distributed to the workers). The domain is monitored and can dynamically grow or shrink as dictated by policies governing that domain. Results of the execution are represented in ways defined by the service layer (e.g., they may or may not be returned to the client directly).

Domains can be used with systems supporting loosely-coupled preemptible processes such as Condor [5], BOINC [4], or Swift [6]. Examples of how an application can dynamically provision resources based on the state of its scheduler queue have been described in [7, 8]. Alternatively, they can also be used with applications requiring stronger synchronization using primitives such as leader election [9] as described in Section 4.2 when discussing our system design.

4. APPROACH

The Domain Manager service implements the concept of a domain, i.e., ensures that it is properly deployed and contextualized, and the number of worker units within a domain scales up and down according to domain policies. The rest of this section describes the architecture of the Domain Manager.

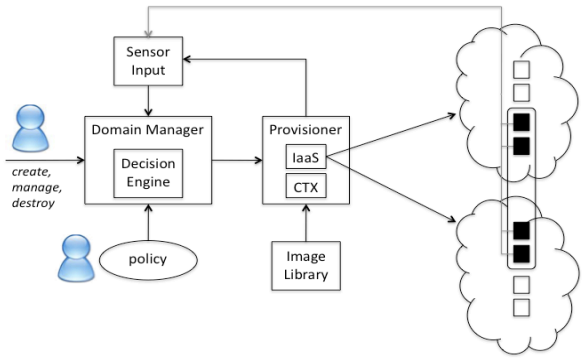


Figure 1: Domain Manager

Figure 1 above shows a domains being managed by the Domain Manager. The domain is distributed over two clouds; with two instances on each. The Domain Manager regulates the size of a domain to reflect sensor input and the associated policies. The Domain Manager relies on sensor information to monitor the deployed VMs, assess their health, and deploy, terminate or redeploy them as needed. The policies are enforced in the context of heartbeat information, informing the Domain Manager of a worker VM health state as well as Provisioner information, informing the Domain Manager of a worker VM lifecycle. Other sources of information can be added. Based on evaluating the information against policies the Domain Manager makes a decision to either add or terminate some of the worker VMs. The Domain Manager decision is carried out by the Provisioner that acts by either deploying or terminating VMs and contextualizing them after deployment them to work as part of the right domain.

To ensure high availability of a domain, all the major components of the system need to be highly available. Furthermore, the system depcted in Figure 1 itself is bootstrapped and monitored by cloudinit.d [10] which ensures repeatable deployment and ongoing monitoring. Note that if any of the system components die, it will not affect the execution of the domain; it will merely affect how quickly it responds to user policies and sensor inputs.

Section 4.1. provides an overview of the interacting components. The design for high availability is descrbed in Section 4.2. Section 4.3 explains implemtation details.

4.1 Components

The *Domain Manager* Service allows clients to create, destroy, and manage domains. Each domain is associated with a Decision Engine that takes two inputs: (1) policy (which can be dynamically modified by a client) and (2) dynamically provided sensor information that can be both system-specific (e.g., information about VM lifecycle and health state) and application-specific (e.g., size of workload for an application). The Decision Engine is a component of the Domain Manager that evaluates the policies against current sensor input and issues commands to the Provisioner to start or stop VM instances. The output of the Domain Manager is a specific directive to deploy specific numbers of VM instances on specific resources. The Domain Manager is implemented to be highly available as per the design described in Section 4.2.

The *Provisioner* provides an adaptation layer for IaaS sites, and contextualizes deployed VMs. Each launch has a unique identifier, supplied by the client, i.e., the Domain Manager, to provide support for idempotency so that a specific request can be retried without launching an additional VM as a result (this functionality is mirrored by IaaS). A launch request also contains a VM type obtained from the Image Library and scheduling constraints such as instance size, the targeted IaaS provider, availability zones, etc. Optionally, a launch request also contains an attribute bag of name/value pairs that can be injected into the VM type to turn it into a customized VM image. The Provisioner is implemented to be highly available as per the design described in Section 4.2.

The *Image Library* allows the Provisioner to dereference a VM type identifier into the necessary VM image compatible with a particular IaaS site. It also currently does the work of interpolating the attribute bag referred to above into a configuration template specific to the deployable type. The Image Library is implemented to be highly available.

Sensors. To function correctly, the system relies on input from a variety of system-specific and application-specific sensors. The default sensors consist of a VM lifecycle sensor and VM agent. The lifecycle sensor is implemented by the Provisioner; it continually queries IaaS and creates notifications of any VM lifecycles stage change (i.e., whether it has been deployed or terminated). In addition the VM agent runs on every VM instance launched via Domain Manager, watches processes and reports heartbeats to the Domain Manager. The lack of heartbeats for too long of a period causes the Domain Manager to consider the VM instance unhealthy (it may in reality be healthy but suffering from a network partition). The VM agent is bootstrapped by the contextualization process [] executed on VM boot.

4.2 Leveraging the Domain Model

To ensure high availability of the overall system, each of the critical components shown in Figure 1, in particular the Domain Manager, Provisioner and Image Library, has to be implemented as a highly available component, capable of scaling up and down and absorbing failure. The challenge of such implementation was that these services do not lend themselves easily to such an implementation: they have a need for internal synchronization, critical sections, and specialized roles. In this section we will describe how we used the domain model to implement those services and at the same time give an example of how the domain

model could be leveraged by other applications seeking to leverage the domain model. We assume that each service is implemented by a set of processes and for simplicity we also assume that each VM has one process, implementing a service function, that is started on boot.

As per the assumptions about worker units (which can appear or disappear at any moment), leveraging the domain model requires us to decompose the function of the service into independent processes that can die or become unavailable at any moment. When they die, they need to be replaced by new service processes that can join the computation automatically. The challenge of the design is therefore the same as the challenge of adapting applications described in Section 3.1; we need to make sure that (1) any given service process can die without affecting the service function and (2) a service process that is newly started to compensate can automatically replace it.

To implement this, we analyzed the functions of services used in our approach and decomposed them into smaller units of work, isolating critical sections. We discovered that "work units" fall roughly into two categories: work that can be executed by reactors, i.e., multiple service process working at the same time (e.g., processing query requests) and work that can be carried out only by actors, i.e., exactly one service process at a time (e.g., critical sections). We then devised a role relay system that allowed us to weave work of the service components back together. According to the characterization above, for each service we defined several roles: one role for the reactor processes, and one for each of the unique work units. The reactors work on units contained in an internal queue whereas the actors work both on reactor tasks and on their specialized function.

We then implemented the service processes such that they can take on any role required by the service. Whenever one of the reactors dies, its work is taken over by the other reactors (and a new reactor is eventually added). Whenever one of the actors dies, we use the requirement that any service process can take on any role, stage a leader election among the reactor processes, and designate a new actor from amongst those processes (a new reactor is eventually added). In this way we ensure that critical service processes are quickly replaced on failure.

4.3 Implementation

Each component service is made up of multiple worker processes which share a common AMQP message queue. Each worker functions as a reactor, pulling messages from the queue and executing them. Messages are service calls specifying a service operation and a set of parameters. The reactor executes the operation and may return a result to the caller. Each operation is idempotent or has known retry semantics.

All service workers are bootstrapped in the same way with the same configuration. Actors are determined by leader elections implemented via Apache ZooKeeper [9]. These elections are participated in by all workers and determine exactly one worker to be promoted to the actor role. If this elected actor later dies, or is partitioned away, it will lose its actor role and another worker will be elected within a predictable bound of time.

This model is used for each of the component services. The Domain Manager has two actor roles. The Decider actor performs regular invocations of the decision engine for each domain. The Doctor monitors the heartbeats received from each deployed node and determines when to mark nodes as unhealthy.

The Provisioner has a single Leader actor that performs queries of IaaS sites and contextualization services. It adjusts node lifecycle states based on the information determined by these queries.

The Image Library has no actor roles; all workers serve as reactors only.

5. Evaluating Time to Scale

To evaluate how reactive our solution is, we tested the time to scale (TTS) of our service. Our focus was on a scenario which represents the worst-case scenario for our system: where potentially multiple clients request modifications for potentially multiple domains by the deployment of a VM; our TTS measured how quickly those aggregate requests can be satisfied. In our experiments this situation was approximated by using one client and single domain. We tested the system on EC2 micro instances. Initially, we create an empty domain and reconfigure it to be composed of 1,850 instances. Each of these instances are requested independently to the IaaS provider with a VM image containing a pre-installed deployment of our VM agent. Each VM is contextualized at boot time to configure the information unique to this domain.

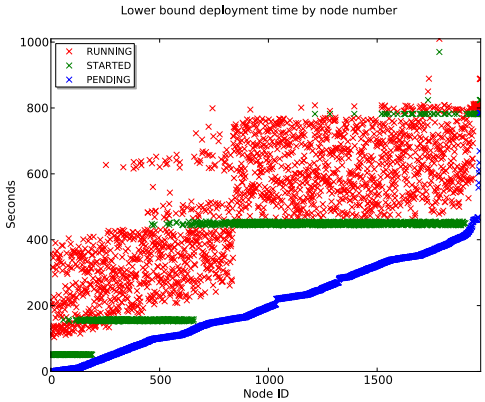


Figure 2: Deployment time by node number

Figure 2 shows that 1,850 instances can be deployed in around 13 minutes. It also shows the various components of the deployment process.

The blue points shows when request to deploy has been acknowledged by Amazon. In our experiments, we found that issuing requests to the IaaS provider can very quickly become a bottleneck. To compensate for that, we deployed a multi-Provisioner, implemented based on the domain model described above; the data shown in the figure was deployed using 10 Provisioner processes. Further, we found that in addition to instance limits (which are explicit and specific to accounts), Amazon also has request limits per time unit which are not explicitly defined and hard to track; however, once such a limit is exceeded, the requests are rejected and the deployment of those VMs needs to be re-requested. One possible way of dealing with it is implementing exponential backoff. Another issue we discovered is that Amazon EC2 micro instances can be unreliable, with a small fraction of instances never reaching a successful VM boot. The strategy we implemented here is overprovisioning, i.e., requesting a certain percentage of VMs above the desired number (in our case it was ~10%) so that a desired number may eventually be obtained.

The green points shows when Amazon deployed the VM, i.e., transferred the image and started it booting. While measuring this quality we discovered that Amazon sometimes does not report transition to this state in a timely fashion; sometimes the reporting is so far from reality in fact that it looks as if the VM has died. For

this reason we approximated the moment when the VM reaches this state by the time it starts contextualization (as reported by the contextualization agent). The pattern visible in the green straight lines in the Figure 2 shows when groups of VMs reached that state and is an artifact of sampling frequency; more frequent sampling of this information would produce more leaning lines.

Finally, the red points show when contextualization for the specific instances was finished. This is the point when the VM is ready for use, i.e., the operating system has booted and all additional configuration work has finished. This group of points has the highest variance as system boot and contextualization take time and phenomena such as noisy neighbour introduce a significant level of variability to how fast processes in the VMs can execute. Another potential issue is sequential context queries, which make some VMs wait.

6. RELATED WORK

Several commercial tools [11-14] provide capabilities covering some subset of the work described here but are either tied to specific commercial provider, specialized for a particular mode of usage not consistent with scientific requirements, or closed proprietary solutions that cannot be studied or adapted to scientific resources. Our purpose is to build a highly adaptable system capable of executing in a multi-cloud environment.

The University of Victoria's Cloud Scheduler project [15, 16] also demonstrated the viability of similar concepts when applied to the scientific community, but with emphasis on sharing resources provisioned in the cloud between different communities using batch queue systems, rather than more general need-based scaling. Cloud Scheduler monitors a Condor queue for new jobs, and provisions resources across Nimbus clouds and Amazon EC2.

Several projects [17-21] evaluate different policies in the context of auto-scaling systems such as the one presented here. Our objectives are focused on the design and implementation of the system itself rather than the policies described in those works.

7. SUMMARY

We described a system that allows users to automatically outsource their computational needs to infrastructure cloud providers. We describe the properties of scalability and availability that we seek to provide in our design as well as the model for applications to use in order to leverage them. Our EC2 evaluation highlights several deployment challenges at infrastructure cloud providers that arise when implementing a system of this type to work at large scales and provides insight into how they were overcome.

8. ACKNOWLEDGMENTS

This material is based on work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed.", in part by the OOI Cyberinfrastructure program funded through the JOI Subaward, JSA 7-11, which is in turn funded by the NSF contract OCE-0418967 with the Consortium for Ocean Leadership, Inc., and in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

9. REFERENCES

[1] Armbrust, M., et al., Above the Clouds: A Berkeley View of Cloud Computing. 2009, University of California at Berkeley.

[2] Amazon Elastic Compute Cloud (Amazon EC2) http://www.amazon.com/ec2.

[3] Jackson, K., L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud Amazon Web Services Cloud. in CloudCom. pp.159-168, Nov. 30 2010-Dec. 3 2010, Indianapolis, IN.

[4] Berkeley Open Infrastructure for Network Computing. 2002; Available from: http://boinc.berkeley.edu.

[5] Litzkow, M.J., M. Livny, and M.W. Mutka, Condor - A Hunter of Idle Workstations, in 8th International Conference on Distributed Computing Systems. 1988. p. 104-111.

[6] The Swift Parallel Scripting Language: http://www.ci.uchicago.edu/swift/main/.

[7] Harutyunyan, A., P. Buncic, T. Freeman, and K. Keahey, Dynamic virtual AliEn Grid sites on Nimbus with CernVM. Journal of Physics: Conference Series, 2010. 219(7).

[8] Marshall, P., K. Keahey, and T. Freeman, Elastic Site: Using Clouds to Elastically Extend Site Resources. CCGrid 2010, 2010.

[9] Hunt, P., M. Konar, F.P. Junqueira, and B. Reed, ZooKeeper: Wait-free Coordination for Internet-Scale Systems, in USENIX Annual Technology Conference 2010.

[10] Bresnahan, J., T. Freeman, D. LaBissoniere, and K. Keahey, Managing Appliance Launches in Infrastructure Clouds. TeraGrid Conference, 2011.

[11] Amazon Web Services: Auto Scaling: http://aws.amazon.com/autoscaling/.

[12] Cloud Foundry. 2011: http://www.cloudfoundry.com/.

[13] RightScale: www.rightscale.org.

[14] SCALR: http://scalr.com/.

[15] Armstrong, P., et al., Cloud Scheduler: a Resource Manager for a Distributed Compute Cloud. 2010.

[16] Sobie, R., et al., Data Intensive High Energy Physics Analysis in Distributed Cloud, in High Performance Computing Symposium. 2011: Montreal, Canada.

[17] Mao, M. and M. Humphrey. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. in SC11. 2011.

[18] Marshall, P., H. Tufo, K. Keahey, D. LaBissoniere, and H.M. Woitaszek. Architecting a Large-Scale Elastic Environment - Recontextualization and Adaptive Cloud Services for Scientific Computing. in ICSOFT. 2012. Rome, Italy.

[19] Marshall, P., H. Tufo, and K. Keahey. Provisioning Policies for Elastic Computing Environments. in 9th High-Performance Grid and Cloud Computing Workshop and the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2012.

[20] Ruth, P., P. McGachey, and D. Xu, VioCluster: Virtualizationfor Dynamic Computational Domains. IEE International Conference on Cluster Computing, 2005.

[21] Assunacao, M.D., A.D. Constanzo, and R. Buyya. Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters. in 18th ACM Symposium on High Performance Distributed Computing. 2009.

Network-Aware Scheduling of MapReduce Framework on Distributed Clusters over High Speed Networks

Praveenkumar Kondikoppa , Chui-Hui Chiu, Cheng Cui, Lin Xue and Seung-Jong Park
Department of Computer Science, Center for Computation & Technology,
Louisiana State University, LA, USA
pkondi1,cchiu1,ccui,lxue2,sjpark@lsu.edu

ABSTRACT

Google's MapReduce has gained significant popularity as a platform for large scale distributed data processing. Hadoop [1] is an open source implementation of MapReduce [11] framework, originally it was developed to operate over single cluster environment and could not be leveraged for distributed data processing across federated clusters. At multiple federated clusters connected with high speed networks, computing resources are provisioned from any of the clusters from the federation. Placement of map tasks close to its data split is critical for performance of Hadoop. In this work, we add network awareness in Hadoop while scheduling the map tasks over federated clusters. We observe 12 % to 15 % reduction of execution time in FIFO and FAIR schedulers of Hadoop for varying workloads.

Keywords

Federated Clouds, Hadoop Scheduling

1. INTRODUCTION

Large scale scientific and engineering applications generate vast amount of data. For example, the Large Hadron Collider project generates peta bytes of data which need to be processed, analyzed and stored. Cloud computing coupled with MapReduce programming paradigm has made huge strides in big data analysis. Hadoop which is an open source implementation of MapReduce is extensively used on compute clouds at Amazon, Yahoo and Rackspace. Many of these commercial clouds provide unlimited computing power. In addition to those commercial clouds, many universities and research laboratories have deployed their own private clouds for research and education.

To maximize the utilization of heterogenous public and private clouds, researchers want to launch their jobs over those clouds at the same time. For example at Louisiana State University, researchers can make use of different cloud resources: LONI [5], FutureGrid [4] and Xsede [8]. This paper focuses on how to aggregate cloud resources from private and public clusters and run MapReduce jobs more efficiently.

Conventional MapReduce(Hadoop) schedulers are optimized to operate over single cluster which connects computing nodes within local networks. However with considerable administrative efforts MapReduce can be deployed over distributed clusters. There are several issues on performance of MapReduce over distributed clusters. For example, network latency and bandwidth between a master node and slave nodes reduce the performance of MapReduce because MapReduce configuration requires a direct communication between a master node and slave nodes directly. Additionally, scheduling tasks among slave nodes which are scattered over remote clusters becomes an important factor on performance because of data locality.

There are several ways of deploying MapReduce Hadoop over distributed clusters. First approach is to set up independent Hadoop clusters at each site, a centralized job scheduler to manage job submission. Second approach is to aggregate underlying physical clusters as a single virtual cluster and run MapReduce tasks on top of this virtual cluster. And the third approach would be to make MapReduce framework work directly with multiple clusters without additional virtual clusters. In this paper, we deploy global MapReduce over federated clusters to utilize the computing effectively. However, deploying MapReduce framework directly on multiple cluster degrades the performance because of failing to exploit data locality. Our research provides a method to add network awareness to global MapReduce so that a scheduler has the information about data locations to launch map tasks at nodes with data.

2. BACKGROUND

2.1 MapReduce Paradigm

MapReduce framework abstracts programmers from the complexities of input data distribution, parallelization, fault tolerance and synchronization of various constructs. The open source implementation of MapReduce framework consists of computing environment called Hadoop and distributed filesystem called Hadoop distributed filesystem (HDFS). Hadoop acts like runtime environment for the Map and Reduce functions provided by programmers. HDFS is designed to provide replication, integrity of the data and fault tolerance. The architecture of Hadoop has master services namely Jobtracker and Namenode and client services namely Tasktrackers and Datanodes. The Jobtracker receives the job submitted by the user and splits it into map and reduce tasks. It

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, September 21, 2012, San Jose, CA, USA.

assigns tasks to Tasktrackers, monitors about task completion status and when all the tasks for a job are complete it reports back to the user.

2.2 Related Works

Cardos et al. [10] suggest different architectures to set up MapReduce framework when source data and computation are distributed. They recommend three architectures, Local MapReduce where in data is moved to centralized cluster to perform computation. Global MapReduce where cluster is established with nodes from all the clusters and Distributed MapReduce where in small independent MapReduce clusters similar to architectural overlay. However, their suggestions are different configurations and fail to address data locality concerns.

Yuano Luo et al. [12] introduce hierarchical MapReduce by organizing the resources into two layers: a top layer consisting of job scheduler and workload manager and a bottom layer consisting of distributed clusters running local MapReduce. When a job is submitted, it is split into sub jobs, which are assigned to distributed clusters. A Jobtracker at each of the clusters reports the job progress and sends result back to a global controller where reduce tasks take place. The disadvantage of this approach is that it is suitable for map-intensive or map-mostly type of applications. If intermediate data generated from the map phase are huge and need to be transferred to a reducer then data transfer becomes bottleneck for the performance of Hadoop. Majority of the data intensive applications involve significant reduce phase computations. Also users need to add additional programming construct to perform global reduce at the global controller.

2.3 Scheduling in Hadoop

2.3.1 FIFO Scheduler

Default scheduling in Hadoop is through FIFO scheduler. The communication between the master node and slave nodes happen through heartbeat packets. When a slave node with empty map slot sends a heartbeat packet to the Jobtracker, scheduler checks the head of the queue job for the tasks. If the job has tasks whose input split is located on the slave node then task is assigned to the Tasktracker. If scheduler does not find local map task then it assigns only one non local map task to this Tasktracker.

2.3.2 FAIR Scheduler

FAIR scheduling ensures that every job gets an equal share of resources over time. If there is single job running then entire cluster is allocated for the job. If new jobs are submitted then task slots which free up are allocated to the newly submitted jobs, so that each job gets roughly equal share of cpu time. Small jobs gets finished within reasonable time and long jobs are not starved. FAIR scheduler organizes jobs into pools and inside pool FIFO or FAIR queuing of the jobs can be configured.

2.4 Global MapReduce over Federated Clusters

MapReduce Hadoop cannot be easily deployed over distributed clusters of different administrative domains to form single MapReduce cluster, since master node services require direct communication between slave node services. If clusters belonging to different administrative domains need to be federated, then considerable administrative effort is needed to make internal nodes of clusters to communicate with each other. We address this issue of internal nodes accessible from outside by introducing a virtualization on top of all the physical clusters. Cloud computing software like Eucalyptus [3] or Openstack [7] is installed on each of the underlying physical clusters. Computing units are provided as virtual machines from any of the distributed clusters. A virtual layer unifies the resources provided by all the distributed clusters into single infrastructure layer of virtual machines. MapReduce cluster is set up on top of these virtual machines where in a single Jobtracker takes the responsibility of scheduling the jobs and managing the Tasktrackers as shown in figure 2. The advantage of this approach is that, it is easy to manage distributed clusters through cloud computing software. However, the critical aspect in this approach is to make a scheduler be topology-aware in order to accomplish data local computation of map tasks. The proposed work focuses on improving data locality while scheduling map tasks for global MapReduce so that all types of data intensive applications such as, map-only, map-mostly and map-reduce application types are supported. In the below sections, we describe data locality issue and propose network aware scheduling to make global MapReduce scheduler data aware.

2.5 Case for Data Locality over Federated Clusters

Unlike HPC systems where computation and file system are decoupled, in MapReduce Hadoop filesystem (Datanode) and computation (Tasktracker) are co-located on a node. When data is transferred to HDFS it is replicated to datanodes so that Tasktracker running on that node will have data blocks prior to execution of a task. In the current implementation of MapReduce Hadooop, data locality is achieved by replicating the data at three levels of cache. At the first level, a data split is stored on a node. At the second level, a data split is saved on a node in the same rack and at the third level, a data split is stored off-rack. To achieve data locality, map tasks are scheduled on nodes which already have the data blocks (this constitutes node local). If it cannot find such a node then tasks are scheduled on nodes which can fetch data blocks from any node in the same rack (this constitutes rack local). If both situations are not met then the map task is scheduled on any tasktracker requesting for a task. Thus MapReduce scheduling algorithms exploiting data locality are based on the assumption of multiple nodes in a rack, which are connected by an aggregate switch.

Figure 1 shows the typical architecture for federated clusters which may be connected by regular internet or by dedicated link. Furthermore, if virtualization is added on top of these federated clusters to provide virtual machines as computing units, then Hadoop's assumption of racks connected through aggregate switch cannot be extended to achieve data locality. To get maximum performance of Hadoop, it is critical to configure Hadoop so that it knows the topology of the entire network while making scheduling decisions.

3. NETWORK AWARE SCHEDULING

The goal of task scheduling in Hadoop is to move computation towards data. If it can't meet this objective then a task is scheduled on a node which is requesting for a task, this causes data to be transferred to compute node for pro-

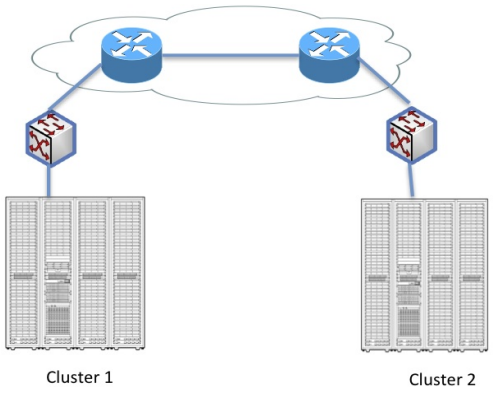


Figure 1: Federated Clusters

cessing. When resources are provisioned by distributed clusters, moving the data across the network causes the degradation in Hadoop's performance.

In case of distributed clusters, the Tasktrackers from any of underlying cluster might request for a map task. The technique to make Hadoop scheduler aware of network topology is to extend the rack aware feature of the existing Hadoop scheduler to provide one more level of caching. An administrator controlled script will hold the information about which cluster the Tasktracker is associated with. In our implementation of global MapReduce architecture, network topology script has information about virtual machines and physical location of the cluster from which they are provisioned. We use Neuca [3] enabled Eucalyptus cloud computing software to provide virtual resources from the distributed clusters. The locations of the virtual machines are organized as $/\text{cluster}_N/\text{rack}_N/\text{vm}_N$. Where cluster_N denotes the physical which cluster, rack_N denotes the rackid and the vm_N indicates the hostname for the virtual machine.

We enable delay scheduling [13] to take maximum use of data locality while scheduling a task. When head of the queue task doesn't find a compute node with data then scheduling of the task is delayed for a specified duration of time. If any of the compute nodes become free with a data split corresponding the job being processed then scheduler assigns a map task to requesting Tasktracker. Duration for which a head of the queue map task is to be delayed is based on the average length of the map tasks for a job hence requires careful tuning.

4. EXPERIMENTAL SETUP

CRON [9] is an Emulab based testbed; a cyberinfrastructure of reconfigurable optical networking environment that provides multiple emulation testbeds operating up to 10Gbps bandwidth. It consists of two main components: (i) hardware (H/W) components, including a switch, optical fibers, network emulators, and the workstations required to physically compose optical paths or function at the ends of these paths; and (ii) software (S/W) components, creating an automatic configuration server that will integrate all the H/W components to create virtual network environments based on the users requirements. All components are connected with 100/1000 Mbps Ethernet links for control. Each workstation is connected with 10 Gbps optical fibers for data movement.

Table 1: Jobs with Varying Task Lengths

Task	Average Execution Time
Map	5 sec
Map	8 sec
Map	14 sec

We use Eucalyptus as cloud computing software, we setup two NEuca-patched [6] Eucalyptus Clouds to construct the distributed cloud scenario. The NEuca patch attaches additional exclusive virtual NICs to a VM for application data transmission. We have two computing nodes in each Cloud and each computing node accommodates three VMs. Then, all VMs connect to each other through the additional NICs to form a virtual cluster. We deploy network aware Hadoop over this virtual cluster as shown in figure 2

We avoid multiplexing VMs on a physical machine for the better and stabler execution. Each VM is allocated with 2 physical CPU cores, 2 GB of physical memory, and 10 GB of local hard disk space. Regarding the NIC, the VirtIO interface is adopted to serve the virtual NIC so that the transmission rate is as high as the Hypervisor can provide.

4.1 Experiments

4.1.1 *Experiments with Native Hadoop*

In this set of experiments we run Hadoop over distributed cluster without network awareness. One of the node will be a master node running Jobtracker and Namenode services. Remaining nodes from both clusters run slave services namely Tasktrackers and Datanodes. We use the wordcount application for evaluation. Input file size for word count application is varied so that average execution time for map task differs. Multiple jobs consisting of tasks with different execution times as shown in the table 1 are submitted.

4.1.2 *Experiments with Network Aware Hadoop*

In this set of experiments, Hadoop with network awareness is configured over the federated cluster. One Jobtracker will manage the scheduling over all the clusters. Tasktrackers from different clusters request for map tasks as and when they get map slots freed. Jobtracker uses the cluster awareness to schedule tasks on these Tasktrackers there by im-

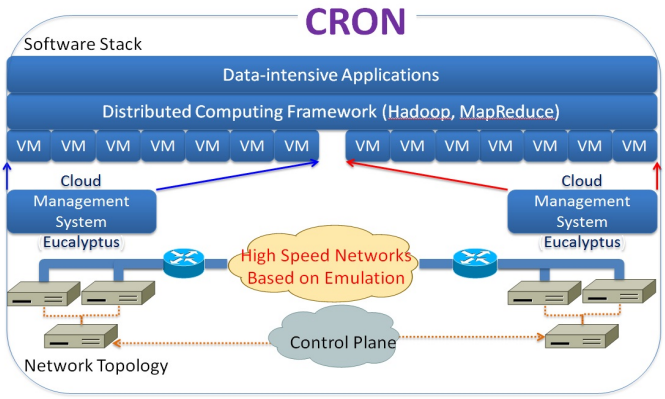


Figure 2: Experimental Setup

Table 2: Evaluation Environment

Nodes	Quantity	Hardware and Hadoop
Master Node	1	2 CPU core, 2 GB RAM, 10Gbps NIC Jobtracker and Namenode
Slave Nodes	12	2 CPU core, 2 GB RAM 10Gbps NIC Tasktracker and Datanode Hadoop-0.20.203.0 2 Map and 2 Reduce Tasks

proving the data locality. We perform experiments with varying inter-cloud bandwidth and delay and measure the performance of Hadoop with network awareness.

5. RESULTS

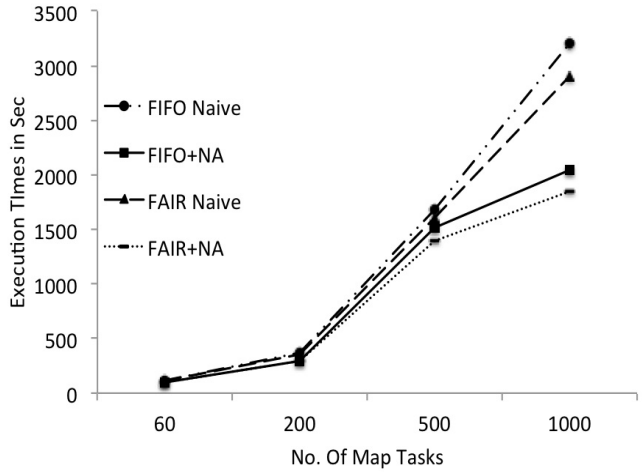


Figure 3: Execution times for Different schedulers for 1Gbps ,30ms inter cloud bandwidth and delay

We compare the execution times of Hadoop native scheduling with network-aware Hadoop for varying delay values. Figure 3 shows the execution times for varying number map tasks, FIFO scheduler with network awareness identified as FIFO+NA, shows the reduction in execution time of about 12 % on average and maximum reduction of 15 % when number of map tasks is higher compared to Hadoop Naive scheduler identified as FIFO Naive. FAIR scheduler with network awareness identified as FAIR+NA also shows the similar results, FAIR scheduler in general takes less execution time compared to FIFO, The reason being effective utilization of the cluster. Network aware Hadoop shows maximum performance improvement for map tasks between 500 to 1000. As the map task number increases, the data splits associated with the map tasks will be spread over the distributed clusters. When Jobtracker has to schedule a map task over a tasktracker, it check if the tasktracker is located in the same cluster as the datanode having the data to process. If this condition is met then only task will be assigned to the tasktracker otherwise it will be delayed for certain delay in seconds specified for the delay scheduling.

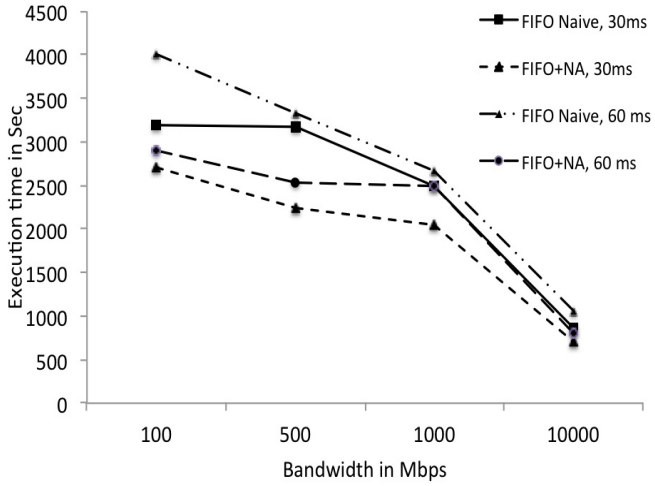


Figure 4: Execution times for varying inter-cloud bandwidth

Bandwidth is a significant factor for federated clusters. The performance of Hadoop greatly depends on the bandwidth between the clusters . Figure 4 is for varying inter-cloud bandwidth with fixed delay. We emulate the varying bandwidth cases on our experimental set up using dummynet [2]. At 100Mbps bandwidth between the clouds, there is significant difference in execution times for processing 10GB data. If the tasks are not executed either with node local or rack local then large amounts of data will be moved from one cluster to another cluster for map task execution. With 100Mbps of bandwidth there will be less available bandwidth between nodes. As the bandwidth provisioned between the clusters is increased overall execution time decreases. However, for processing tera bytes of data there should be sufficient bandwidth provisioned between the clusters. We performed experiments with different network conditions, in all the cases significant reduction in execution time is observed between native Hadoop scheduler and network-aware scheduler as the number of map tasks increased. It is evident that for higher inter cloud bandwidth the performance of Hadoop with network aware is greatly increased compared to native Hadoop scheduling.

Figure 5 shows percentage increase in the local map tasks. As explained in the earlier sections, Hadoop scheduling follows the philosophy of moving the computation to data. All the tasks which have the data on the same node as the one requesting for the tasks are scheduled first, this is referred as node local. If scheduler cannot find such tasks then all the tasks whose data is located in the same rack as the node requesting for the task are scheduled, this is referred as rack local. Network awareness is applied to non local map tasks which require to fetch data from some other data nodes. Network aware Hadoop minimizes the data movement from one cluster to another while executing map task by adding cluster level locality. We enable delay scheduling to optimize the data locality. Delay scheduling ensures that before the task is scheduled on a tasktracker which does not have the data to process will be skipped for configured amount of time. If any of the tasktracker becomes free in that dura-

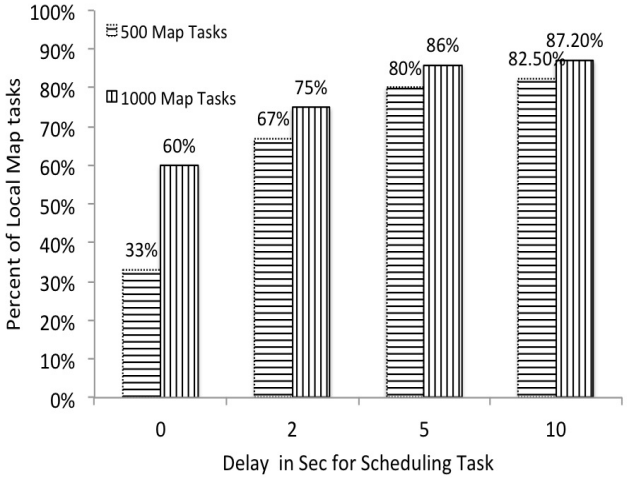


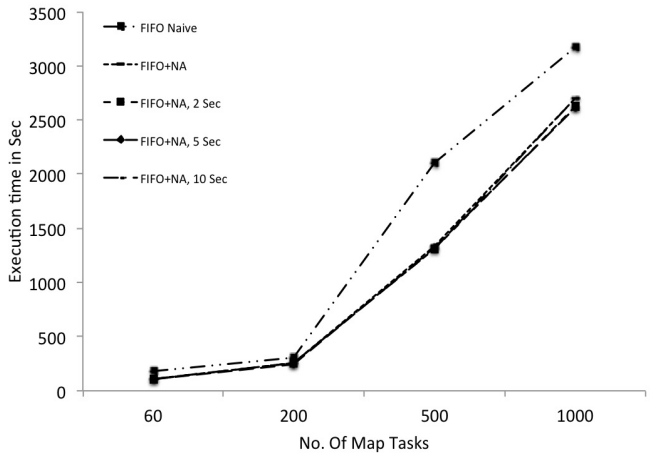
Figure 5: Increase in local map tasks with Network-aware

tion which as the data to process then, task is scheduled on the second tasktracker. For data intensive applications, data split movement takes more time than processing of the data split. Delay parameter is a sensitive parameter and should be carefully configured. In the Figure 5, delay of 0 sec refers scheduling without enabling the delay scheduler. With 2 Sec delay, non local map tasks will be delayed for 2 sec to check if any of the other tasktrackers request for a task. Since the length of each task is small, it is observed that tasktrackers get freed very frequently and request for a task. It is for this reason the delay parameter depends of the length of the task execution. These results show that network awareness coupled with delay scheduling could be used to minimize the transfer of the data between the clouds.

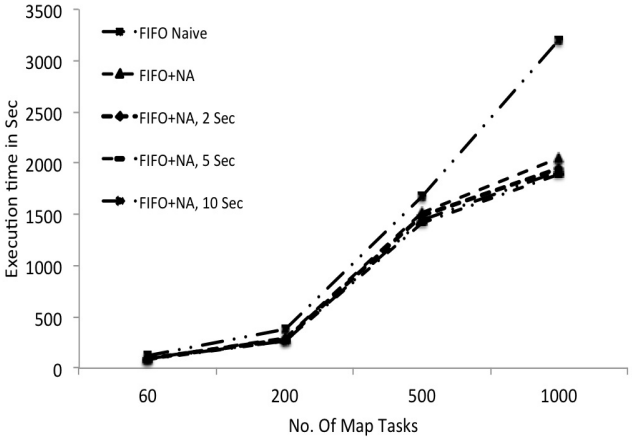
Figure 6 shows the results for 30ms inter cloud delay for varying bandwidths of 100Mbps and 1 Gbps. For the same number of map tasks, the execution times for a Job is less for 1Gbps bandwidth. In case of 100Mbps bandwidth the gap between curves for different scheduling schemes is less significant. Reduced available bandwidth between the clusters causes bottleneck for data split movement for tasks scheduled on distributed clusters. In figure 6(a), for tasks ranging from 500 to 1000 the execution times for native Hadoop and different variants for Network aware Hadoop, significant difference is observed though the inter cluster bandwidth is less.

Figure 6(b) shows experimental results for 1Gbps inter cloud bandwidth and 30 ms delay. Since higher bandwidth is provisioned between the clusters the overall execution times for the jobs compared to figure 6(a) is less. Further for jobs with higher number of tasks i.e tasks ranging from 500 to 1000 map tasks, the performance of network aware Hadoop is significantly improved. This shows that the performance Hadoop could be increased with provisioning higher bandwidth, less delay coupled with locality awareness. Similar results were observed for experiments conducted with fair scheduler.

Figure 7 shows 60ms inter cloud delay and bandwidths of 100 Mbps and 1 Gbps. There is significant reduction in the overall job execution times between 100 Mbps and 1 Gbps scenarios. However, figure 7(a) shows results for 100 Mbps



(a) 100Mbps bandwidth and 30ms delay between clusters



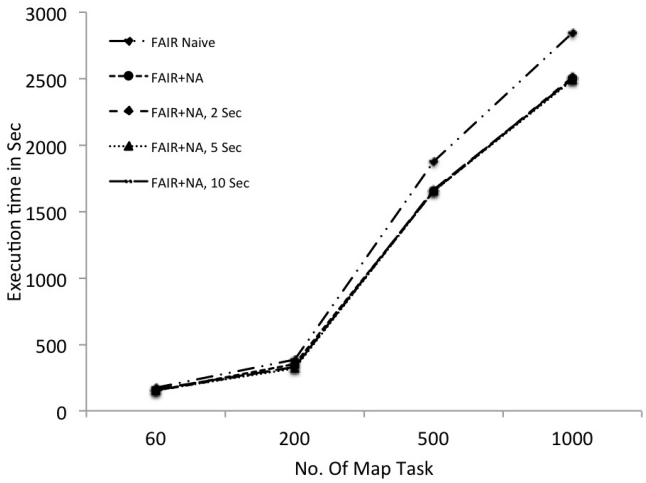
(b) 1Gbps bandwidth and 30ms delay between clusters

Figure 6: Execution times for 30ms inter-cloud delay

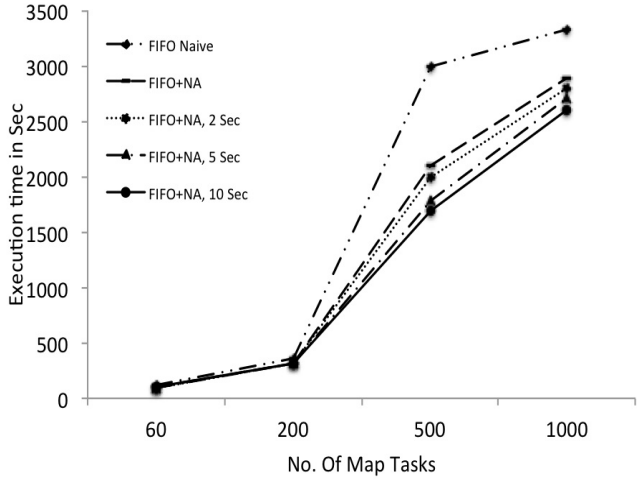
and 60 ms inter cloud network parameters. Because of the higher delay between the clusters, there is not significant improvement between the native Hadoop and the different variants of network aware Hadoop. Figure 7(b) is for 1 Gbps inter cloud bandwidth and 60 ms delay. In this case also because of higher delay between the federated clusters, less performance improvement is observed. However, it is better compared with the 100Mbps scenario.

5.1 Further Discussion

Figure 8 shows results for 10Gbps bandwidth and 30 ms delay between clusters. We observe that for higher bandwidth and low latency, overall execution times are greatly reduced for both native Hadoop and network aware Hadoop. However it is expected that network aware Hadoop to perform significantly better for higher bandwidth and low latency but significant difference is not seen from the results shown in the graph. The cause of this behavior is that the virtual interfaces of VM's cannot provide throughput more than 1.2 Gbps. Even though available bandwidth is 10Gbps,



(a) 100Mbps bandwidth and 60ms delay between clusters



(b) 1Gbps bandwidth and 60ms delay between clusters

Figure 7: Execution times for 60ms inter-cloud delay

the utilization from each of VM does not exceed 1.2 Gbps.

6. CONCLUSIONS AND FUTURE WORK

Placement of the map task on a node which has the data to process is critical for the performance of Hadoop over federated clusters. Single-cluster MapReduce architecture may not be suitable for situations when data and compute resources are widely distributed. In this work, we provide network awareness to the FIFO and FAIR schedulers in Hadoop. We evaluate our implementation on resources provided by CRON testbed. Performance improvement of 12 % to 15 % is observed in both FIFO and Fair schedulers. We plan to extend the network awareness while placing the reduce task since Reduce phase in MapReduce adds up significantly to the overall execution time.

Acknowledgement: This work has been supported in part by the NSF MRI Grant #0821741 (CRON project), GENI grant (BBN,NSF) and DEPSCoR project N0014-08-1-0856.

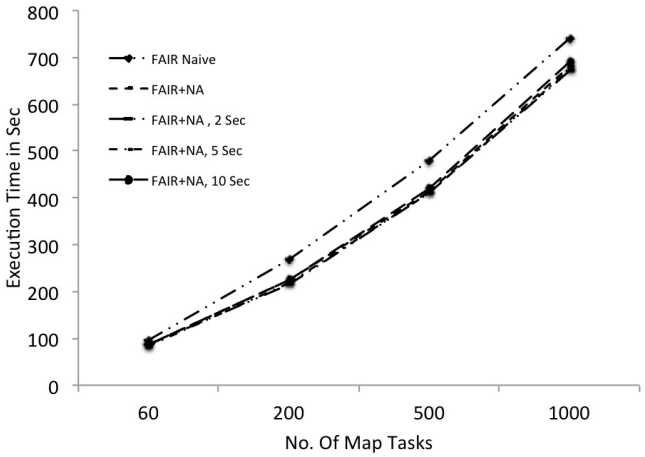


Figure 8: Execution times for 10Gbps and 30ms between clusters

7. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Dummynet software emulator. http://info.iet.unipi.it/luigi/dummynet/.

[3] Eucalyptus Cloud Computing Software. http://www.eucalyptus.com/.

[4] Guture Grid Project. http://www.futuregrid.org/.

[5] Louisiana Optical Network Initiative. http://www.loni.org/.

[6] Neuca Eucalyptus Cloud Computing Software. https://geni-orca.renci.org/trac/wiki/Eucalyptus-2.0-Setup/.

[7] Openstack Cloud Computing Software. http://www.openstack.org/.

[8] XSEDE: Extreme Science and Engineering Deicovery Environment. http://www.futuregrid.org/.

[9] CRON Project: Cyberinfrastructure for Reconfigurable Optical Networking Environment, 2011. http://www.cron.loni.org/.

[10] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman. Exploring mapreduce efficiency with highly-distributed data. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004.

[12] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W. W. Li. A hierarchical framework for cross-domain mapreduce execution. In *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, ECMLS '11, pages 15–22, New York, NY, USA, 2011. ACM.

[13] M. Zaharia, K. Elmeleegy, D. Borthakur, S. Shenker, J. S. Sarma, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *In Proc. EuroSys*, 2010.

Federated Cloud-based Big Data Platform in Telecommunications

Chao Deng
China Mobile Research Institute
Beijing 100053, China
dengchao@chinamobile.com

Ling Qian
China Mobile Research Institute
Beijing 100053, China
qianling@chinamobile.com

Meng Xu
China Mobile Research Institute
Beijing 100053, China
qianling@chinamobile.com

Yujian Du
China Mobile Research Institute
Beijing 100053, China
duyujian@chinamobile.com

Zhiguo Luo
China Mobile Research Institute
Beijing 100053, China
luozhiguo@chinamobile.com

Shaoling Sun
China Mobile Research Institute
Beijing 100053, China
sunshaoling@chinamobile.com

ABSTRACT
China Mobile is the biggest telecommunication operator in the world, with more than 600 million customers and an ever increasing information technology (IT). To provide better service to 600 million customers and reduce the cost of IT systems, China Mobile adopted a centralized IT strategy based on cloud computing. The big data issue becomes the most significant challenge to the cloud computing based China Mobile IT structure. This paper presents the China Mobile's big data platform based on the cloud. This platform integrates the big data storage, the development and deployment of big data ETL (Extract, Transfer, Load) and DM (Data Mining) into a unified framework. This big data analysis platform can effectively support the analytical tasks in telecommunications, but it can also help China Mobile provide public cloud computing service. In this paper, we introduce the detailed architecture of China Mobile's platform and discuss its performance.

Categories and Subject Descriptors
C.2.4 [**Distributed Systems**]: Distributed applications.

General Terms
Algorithms, Performance, Design, Experimentation

Keywords
Telecommunication; big data; Cloud Computing; Hadoop; SaaS

1. INTRODUCTION
China Mobile Communication Corporation (CMCC) takes a hierarchical structure based on branches. In fact, the China Mobile has 31 branches, where a province is an individual operator branch, and the 31 branches constructed IT infrastructure and applications individually. The 31 branches upload their business data to the headquarters by various periods.

For data-intensive business companies, the increase of user scale and the rising of business applications have brought huge challenges to the IT infrastructure and information processing system. For example, the storage volume of Service Support system at CMCC has reached 8000TB at the beginning of 2012, and the collected data for business analysis reached additional 7000TB. In CMCC, a middle level branch has more than 20 million users, and its CDR (Calling Detail Record) can be 12~16TB in 12 months, and its signal data can be 1TB per day. To provide efficient decision-making proposal, reporting, the OLAP data, and the marketing strategy, the traditional data analysis systems confront both the storage and the computing bottlenecks. The traditional data analysis systems commonly are constructed in a centralized infrastructure based on a few expensive UNIX servers, and then many analysis applications are developed using database or data warehouse products installed on UNIX servers. However, the centralized infrastructure results in low scalability and high cost, due to the massive data to be stored and analyzed. For example, a popular commercial business intelligence system uses clustering algorithms that can only support 1 million user's data for knowledge discovery processing, which lags significantly behind real demand.

China Mobile has total more than 600 million customers. To provide better service to the increasing number of customers within the 31 branches results in great challenges to updating and expanding the IT infrastructure in the branches. Hence, China Mobile adopted a centralized IT strategy based on federated cloud infrastructure to reduce the cost of IT systems. China Mobile has selected 3 provinces with beneficial climate to build the big data centers, and selected centers in 2 provinces as their operations centers. The big data issue becomes the most significant challenge after the centralized strategy is performed by federated cloud infrastructure. The federated cloud consists of large numbers of X86 PC server-based clusters. To reduce the cost of developing various applications on big data, it is required for China Mobile to develop and deploy a unified data analysis platform in the operations centers utilizing federated cloud strategies.

Cloud computing is an emerging technology to help data-intensive companies deal with the large-scale data analysis requirement with low cost and high performance. Cloud computing provides IT supporting infrastructure with flexible scalability, large-scale storage and high performance computing. A typical and successful cloud computing use is demonstrated by Google.

Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, , September 21 2012, San Jose, CA, USA.

Nowadays, the commercial PC cluster along with the distributed parallel computing environment is widely accepted as a popular solution to implementing the low-cost and high performance target. The combination of Google File System (GFS) [1] and MapReduce programming framework represents the necessary synergy between data distribution and parallel computation. Hadoop [2] is an open source implementation of MapReduce and GFS. Hadoop has been widely selected as an infrastructure to provide cloud computing service, such as Yahoo [3], Cloudera [4] and Taobao [5]. Many large-scale data analysis systems also turn to Hadoop and MapReduce, since Google successfully implemented and demonstrated data analysis applications based on MapReduce [6][7]. The recent Gartner report [8] indicates that the Hadoop-based large-scale data analysis solution is disrupting the traditional data analysis landscape. The open source project Mahout [9] provides many MapReduce style data mining algorithms based on Hadoop. Some data warehouse and BI products support data collection, processing and storage with HDFS (Hadoop distributed file system), such as Teradata [10] and Pentaho [11].

To reach the goal of low-cost and high effectiveness operation, China Mobile Research Institute (CMRI) has started the cloud computing project, named *Big Cloud*. Big Cloud includes a series of systems, supporting IaaS, PaaS and SaaS. The infrastructure of Big Cloud is based on commercial PC server cluster and Hadoop platform. BC-PDM (Big Cloud – Parallel Data Mining) is one of the core systems in SaaS service layer of Big Cloud. There are many successful data analysis applications of CMCC developed on the BC-PDM system.

2. BC-PDM

The detailed architecture, functionality, features and performance of BC-PDM are introduced in this section.

2.1 Architecture of BC-PDM

The architecture of BC-PDM system is depicted in Figure 1. We distinguish three layers: the cloud computing cluster layer, the BC-PDM service layer, and large-scale data analysis applications layer.

The role of each layer is explained as follows:

1) The Cloud computing platform layer is built on the X86 PC servers cluster. Hadoop is deployed among these PCs to provide the HDFS, MapReduce, and SQL JDBC support. HDFS can provide an efficient and reliable distributed data storage as well as file control service needed by applications involving large databases. MapReduce is an easy-to-use parallel programming paradigm suitable for large scale, data intensive parallel computing applications, which at the same time offers load balancing and fault tolerance. Hive, HBase and HugeTable offers online ad hoc querying through the JDBC API.

2) The BC-PDM layer provides the parallel data analysis capabilities. Data load and export module supports variety of data sources, such as URL, database, and file systems. Data Exchange Transfer and Management module supports remote file transfer, access with data in databases, and data management remote access control. Parallel ETL and parallel DM modules consist of various large-scale data refining and mining algorithms with MapReduce parallelization. Parallel

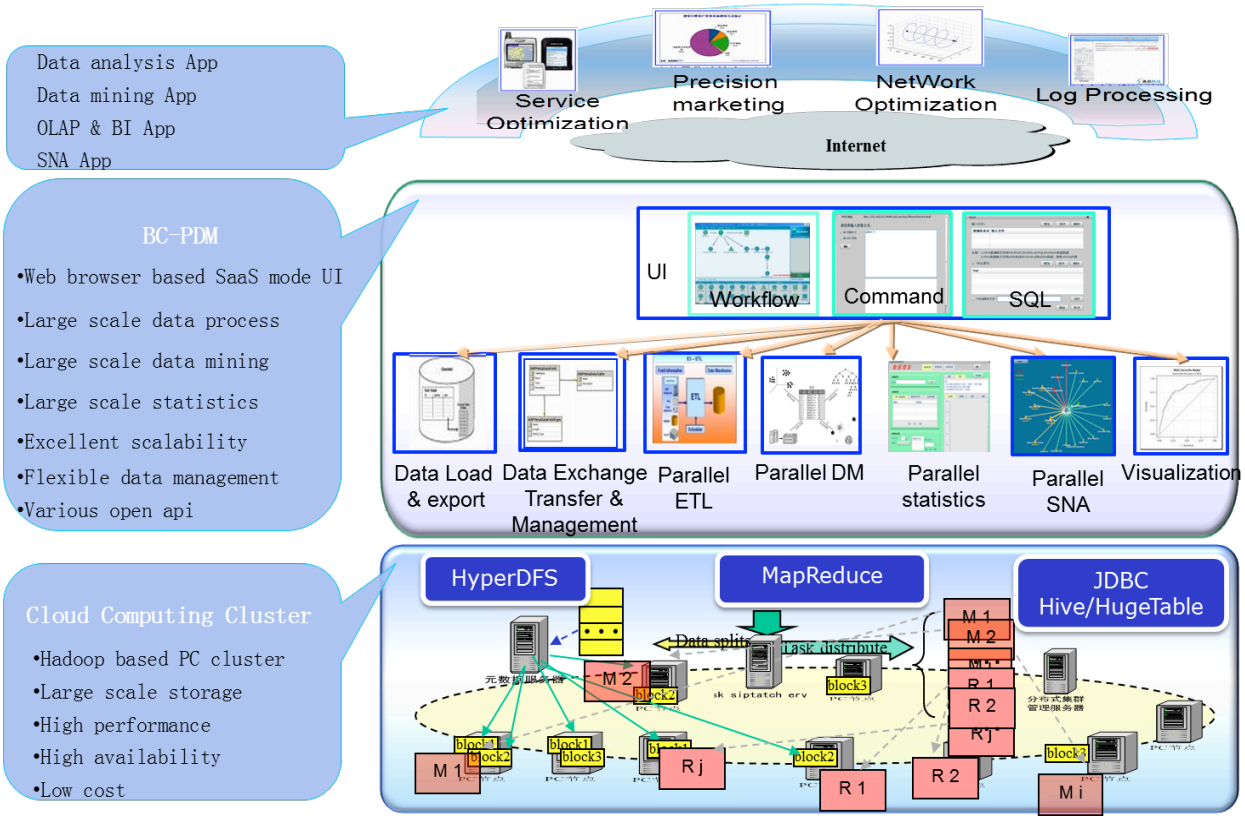


Figure 1. BC-PDM architecture.

statistics module provides large-scale data exploration operation. Parallel SNA module provides large-scale social network analysis algorithms with MapReduce parallelization. Visualization module presenting analysis results to users. A UI module offers workflow, command line, SQL GUI interface to users.

3) The Data analysis application layer consists of various ETL, OLAP, BI, Data mining and SNA applications. All these applications could be developed on the BC-PDM.

2.2 Functionality of BC-PDM

To support various large-scale data analysis requirements, the current version of BC-PDM has provided the parallel ETL, data mining, statistics, SNA, and SQL operations. These capabilities covered almost all popular operations used in our large-scale data analysis.

The Parallel ETL provides 7 classes and 45 operations for data extraction and transformation on large-scale data. The 7 classes are cleaning, transfer, count, aggregation, sampling, sets and update class. The cleaning class includes type-checking, primary key and foreign key checking, missing value checking, duplication and max-min value delete. The transfer class includes case-when, internalization, type-change, normalization and anti-normalization, ID-add, field exchange, and PCA. The count class includes column-generation, group-by and basic statistics. The aggregation class includes delete, join, dimension-table join, sort, where and select. The sampling class includes hierarchical sampling, random sampling and dataset split. The sets class includes sets-difference and sets-union. The update class includes normal-update and insert-update.

The parallel data mining algorithms provides 3 classes and 12 algorithms popularly used in data analysis. The 3 classes are classification, clustering and association. The classification algorithms include C45 decision tree, K-NN, NaiveBayes classifier, BP Neural Network, LR logistic regression. The clustering algorithms include K-means, DBSCAN and Clara. The association algorithms include Apriori, FPGrowth, Awfits and sequence association.

The Parallel SNA provides social network detection of individual node and groups. Furthermore, it can also provide group evolvement tracking.

In addition, the Parallel SQL receives user's SQL shell input, and then submits these SQL command to Hive, HBase or HugeTable to execute through JDBC. Thus the original application developed by SQL can easily be adapted to the BC-PDM.

2.3 Features of BC-PDM

As a successful practice of cloud computing based data analysis system, the BC-PDM has several attractive features:

1) Web2.0 based SaaS (Software as a Service) mode: allow user to connect to internet and develop data analysis applications on Web-browser. Thus, the user doesn't need to buy analysis software or install.

2) MapReduce based parallel algorithms: All the ETL data processing and mining algorithms are developed via MapReduce mode. Thus, the large-scale data computing performance is offered.

3) The PC server cluster based infrastructure: all the storage and computing are based on the commercial PC server. Thus the low-cost and high scalability target are reached.

4) Open architecture: the BC-PDM software defines open architecture that allows the user to develop new algorithm component and add it to BC-PDM. Thus, the user can easily act as a new capability contributor.

5) Various API: each parallel algorithm in the BC-PDM provides Java api, WebService API, and Command line API. Thus the third-party providers can develop their own applications via various modes. In addition, the various api also provide a good selection of implementing PaaS platform.

6) Flexible data management: The data exchange, transfer and management in BC-PDM can help user transfer data from remote to HDFS, load data from DFS into user directory of BC-PDM, and remote control the access of private data and meta information.

7) Workflow and scheduler: Allow user design analysis application by drag and drop operation. The workflows between different users could be shared and reused with authority. The scheduler helps user assign existing applications to run under predefined time and conditions.

8) CWM and PMML output: the ETL operations and data mining models in BC-PDM are exported as CWM standards and PMML description. Thus the process and model from BC-PDM can be reused in other systems that support the CWM and PMML.

2.4 Performance of BC-PDM

To evaluate the scalability of BC-PDM for large-scale data analysis, the performance of ETL and mining algorithms on PC server cluster is tested. The number of Hadoop based PC nodes is increased from 32 to 64 and 128. And the experimental results are shown in Figure 2 and Figure 3. The experiment results indicate that the ETL processing has excellent scalability; meanwhile, the parallel data mining algorithm has acceptable scalability.

As shown in Figure 2, there are 11 ETL operations with good scalability, whose speedup ratio increases nearly linearly with the number of nodes.

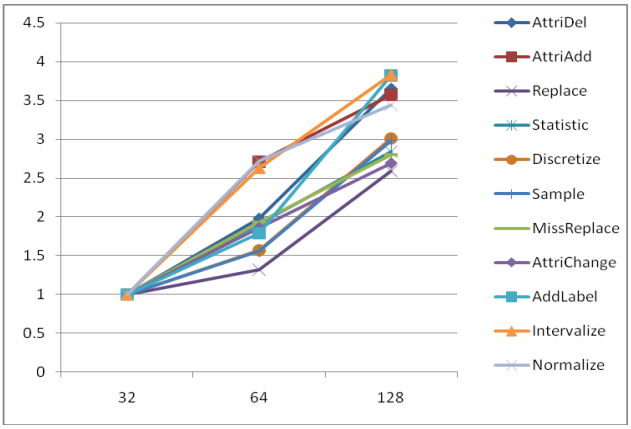


Figure 2. Scalability of ETL operations in BC-PDM.

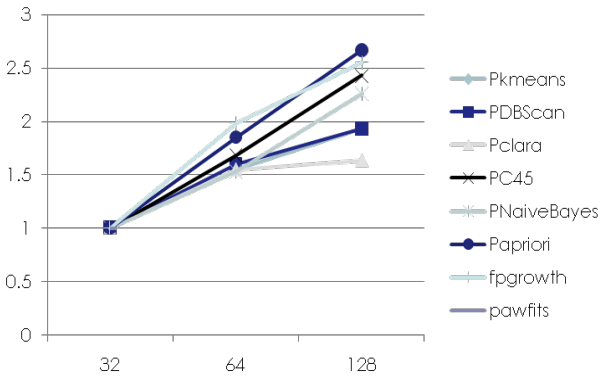


Figure 3. Scalability of mining algorithms in BC-PDM.

Figure 3 indicates that some data mining algorithms (i.e. FP-growth, PC45 and PNaiveBayes) achieved desired scalability; other algorithms also have suitable scalability.

2.5 Performance of China Mobile ETL Application

China Mobile's traditional Business Analysis Support System (BASS) based on the Data Warehouse, and the ETL applications occupy more IT resources than other tasks. Therefore, the ETL applications become a significant factor for increasing the IT cost of BASS. In order to reduce the Data Warehouse burden from ETL applications, we have used BC-PDM to complete the ETL tasks. The GPRS records aggregation is a common ETL application in the China Mobile BASS Data Warehouse. With the fast development of mobile internet and smart phone in China, the GPRS records are significantly increasing. In one China Mobile branch, the task to complete the aggregation of GPRS records of one month includes that the ETL application should group 2TB record data by user id, plan type, access type and brand type dimensions, and then summarize. To output and report the summation value by the end of each month, the traditional application based on our Data Warehouse needs to run 1~2 hours per day to get one day's value, thus the total time is 30 hours for aggregating one month GPRS records. When we use Groupby and Join components to form the GPRS aggregation application on BC-PDM, which is deployed on 10 PC servers, the total time for one month's TB data is reduced to 2 hours 5 minutes. The difference between traditional 30 hours and BC-PDM's 2 hours shows BC-PDM could significantly improve the performance of practical ETL application in China Mobile BASS. As well as the cost of 10 PC servers used by BC-PDM is only 1/10 of the cost of traditional Data Warehouse and Unix servers.

3. USAGE OF BC-PDM

China Mobile is planning to use BC-PDM provide public cloud computing service.

3.1 Basic Usage of BC-PDM

A snapshot of the web browser based workflow GUI using the mode BC-PDM is shown in Figure 4. Our workflow GUI allows the designing of data analysis application via a graphical layout by allowing drag-and-drop of various operation components. The GUI consists of three functional layouts.

The top area is component bar, and all the parallel ETL, parallel data mining, parallel statistics, visualization, and data management capabilities of BC-PDM are exhibited as components for user selection.

The middle layout is application design area, and the user can drag the desired operation from the component bar, configure parameters, concatenate with other components, and form the workflow consistent with the business process. Adittionally, an SQL shell and command shell are allowed to be included in order to input and combine into the workflow as parallel execution threads. Rhrough this mechanism efficiency can be improved and the user can integrate schedulers for the execution of the workflows.

The bottom layout is the state monitor window. The Hadoop cluster state, such as nodes number, HDFS data volume, and nodes logs, are shown. The task progress and MapReduce progress of each job submitted to Hadoop are also captured and

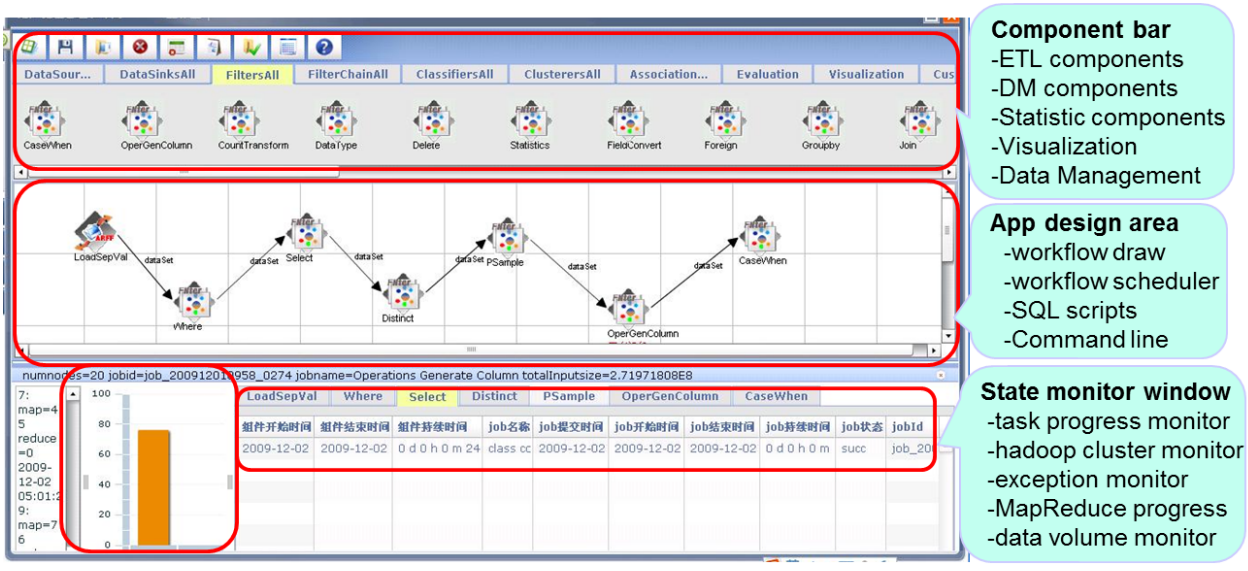


Figure 4. BC-PDM workflow GUI.

shown in the window. Furthermore, exceptions are caught and shown, too.

3.2 Steps in Using Big Cloud by a Public User

A public user should first register as a user of Big Cloud, and then the BC-PDM system could be accessed and authorized. The basic steps of a public registration using BC-PDM system include:

1) Prepare the data at the user-end
2) Download and install the ftp-client provided by BC-PDM at the user-end
3) Upload the prepared data to remote HDFS by using the ftp client, and the data will be stored in the user directory of BC-PDM system.
4) Use web browser to design data analysis application at BC-PDM for data processing and mining
5) Check the result at BC-PDM, and download the result to user-end via the ftp client

4. SUMMARY AND FUTURE WORK

The BC-PDM platform has successfully been used for large-scale data analysis applications with MapReduce and HDFS. Analysis applications in telecommunication industries are a preliminary proof that BC-PDM is a promising schema of helping data-intensive problems to be solved with low-cost, high performance and high scalability IT target. To broaden the usage of data analysis systems based on the cloud computing infrastructure, the future work in BC-PDM focuses not only on the more functionality features and open issues, but also on a user incentives plan.

The new functionality features of BC-PDM will include complex statistics operations, web mining needed operations, OLAP needed operations, reporting service and BI platform service. The open issues that need to be urgently resolved include the hard disk error and I/O exceptions of HDFS PC server cluster that we observed, and the MapReduce limitation on improving the parallel performance of iterative data mining algorithms.

The user incentives plan includes: 1) collect large-scale public datasets, such as Google N-Gram datasets and human gene datasets, and offer free access and free data processing for registered public user; 2) originate data analysis competition with prize at the SDN (software developer network) community.

5. ACKNOWLEDGMENTS

We thank Bill Huang for his guidance and inspiration, and Dejan Milojicic, from HP Labs, for his help in improving the quality of the presentation of the paper.

6. REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system” in In Proceedings of 19th ACM Symposium on Operating Systems Principles, October 2003. LOCATION MISSING, DATE MISSING

[2] Hadoop, an open source implementing of MapReduce and GFS, http://hadoop.apache.org.

[3] Hadoop at Yahoo! http://developer.yahoo.com/hadoop/

[4] Cloudera services on Hadoop. http://www.cloudera.com/hadoop/

[5] Hadoop at tbdata.org. http://www.tbdata.org/archives/category/cloud-computing/hadoop

[6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in In Proceedings of OSDI’04: Sixth Symposium on Operating System Design and Implementation, December 2004.

[7] Sridhar Ramaswamy, “Extreming Data Mining”, Google Keynote speech in SIGMOD 2008. LOCATION MISSING. DATE MISSING

[8] Gartner report. Hadoop and MapReduce: Big Data Analytics. http://www.gartner.com/technology/media-products/reprints/cloudera/vol1/article1/article1.html

[9] Mahout, open source project on data mining algorithms based MapReduce, http://lucene.apache.org/mahout/.

[10] Teradata exchanges with Hadoop. http://developer.teradata.com/tag/hadoop.

[11] Pentaho for Hadoop. http://www.pentaho.com/products/hadoop/.

Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers

Mahendra Kutare
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
imax@cc.gatech.edu

Greg Eisenhauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
eisen@cc.gatech.edu

Chengwei Wang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
wangcw@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
schwan@cc.gatech.edu

Vanish Talwar
HP Labs
1501 Page Mill Road
Palo Alto, CA 94304 USA
vanish.talwar@hp.com

Matthew Wolf
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
mwolf@cc.gatech.edu

ABSTRACT

To effectively manage large-scale data centers and utility clouds, operators must understand current system and application behaviors. This requires continuous monitoring along with online analysis of the data captured by the monitoring system. As a result, there is a need to move to systems in which both tasks can be performed in an integrated fashion, thereby better able to drive online system management. Coining the term 'monalytics' to refer to the combined monitoring and analysis systems used for managing large-scale data center systems, this paper articulates principles for monalytics systems, describes software approaches for implementing them, and provides experimental evaluations justifying principles and implementation approach. Specific technical contributions include consideration of scalability across both 'space' and 'time', the ability to dynamically deploy and adjust monalytics functionality at multiple levels of abstraction in target systems, and the capability to operate across the range of application to hypervisor layers present in large-scale data center or cloud computing systems. Our monalytics implementation targets virtualized systems and cloud infrastructures, via the integration of its functionality into the Xen hypervisor.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.4.4 [**Communication Management**]: Network Communications

ICAC'10, June 7–11, 2010, Washington, DC, USA.

General Terms

Design, Measurement

Keywords

Monitoring, Management

1. INTRODUCTION

Performance and program monitoring are well-established areas of research. Traditionally concerned with performance debugging [23], a more recent focus of online monitoring has been to help deal with the ever-increasing complexity and scale of modern IT and data center facilities. Here, monitoring is used to continually measure and assess current system or application behaviors [18], to detect and diagnose problems, or even for purposes of business analytics. The data used by such analyses is captured by a wide variety of tools, operating in specific subsystems, running at different levels of abstraction [14], on entire parallel machines [22, 23], or addressing the distributed nature of underlying infrastructures [19, 21, 26]. These data collection and dissemination tools may be integrated with problem diagnosis systems [20], with program development systems (to better understand the applications being monitored) [22], and they may themselves be managed to improve how monitoring is performed [16]. In most such settings, however, the primary purpose of monitoring is to improve the ways in which systems and applications operate, measured in terms of performance, reliability, power usage, the ability to meet service level agreements (SLAs), and similar metrics important to applications and IT infrastructure providers [17]

Monitoring to Manage Large-scale Systems..

Our research is developing methods and infrastructures to improve the manageability of future data center systems [17]. This paper is focused on a necessary element of system management, which is efficient and scalable online system monitoring. Concerning scale, recent reports indicate that already, up to 25% of enterprise data today is from systems monitoring, with almost 240 terabytes produced annually,

and this number is only going to increase with next generation facilities. Furthermore, scale goes beyond raw system size in that one also has to take into account the multiple time and length scales at which different system components and levels of abstraction operate. Concerning the 'length' scale, consider operator queries about current data center health for large-scale systems vs. providing detailed information about the state of a specific disk subsystem, for instance. Concerning time scales, consider the high rate at which web requests are received and serviced by system-level threads running across multiple processors, compared with the lower rate at which the virtual machines running these threads migrate across machines when being consolidated; or consider the management done to differentiate service levels for (high rate) disk requests vs. the relatively low-rate power management actions applied to the processors that run such disk-centric applications

The examples described above illustrate that when the purpose of monitoring is to better manage systems or applications, monitoring must operate across multiple time scales, across different size systems, and at multiple levels of abstraction – from application-centric entities like 'requests' to infrastructure-centric entities like blades or racks. Furthermore, monitoring must go beyond data capture and dissemination to also understanding and analyzing captured data [16], in addition to support intelligent problem determination methods [11], as needed by subsequent management actions.

Monalytics..

We refer to our combined monitoring and analysis system as 'monalytics'. This paper identifies and explores several important properties of 'monalytics' that are key to its applicability to large-scale IT infrastructures.

Data-local analysis – for low latency response, that is, in order to limit the delays between when monitoring data is first captured to when interesting insights are derived from that data, it must be possible to perform select analyses 'close' to data sources. While traditional systems have eschewed such solutions due to the potential perturbation caused by additional monitoring loads [23], given that modern machines are increasingly bound in performance by memory bandwidth rather than CPU speed, we advocate an approach in which data-near analysis is used both to reduce monitoring data volume and to rapidly gain insights from captured data. In other words, we posit that it is often cheaper to quickly analyze data and then send out data summaries or abstracts than it is to copy raw captured data items – this fact has also been shown to hold for data dissemination across networked machines [7], and we note that for the same reasons, physical systems using smart sensors are increasingly common. For monitoring, this implies the need to combine the flexible data capture done in traditional monitoring systems [23] with low overhead methods for associating light-weight analysis methods with capture mechanisms.

Operation at multiple time scales – it is well-known that monitoring rates (e.g., sampling rates) depend on the artifacts and behaviors being watched, as do window sizes (e.g., sample sizes). Further, systems exhibit multi-time scale behaviors for different hardware sensors (e.g., slowly changing thermal sensors vs. rapidly changing cache miss rates) as well as for instrumented software. As a result, adjustable monitoring rates, window sizes, and associated noise reduction and data filtering constitute the base functionality required by any monalytics system. A corollary is the need to analyze data across multiple time scales, which for analytics, typically requires analysis-specific methods, as when trying to correlate observed temperature changes with changes in IT loads and/or processor utilization. Our monalytics infrastructure makes it easy to adjust rates and window sizes, and to associate filtering and data smoothing codes with capture mechanisms. Also shown in this paper is a useful multi-scale analysis technique, using entropy-based methods.

Scalability to different system sizes – as with the 'scope in time' implied by window sizes, monitoring must also use 'scope in space', meaning that it must be possible to limit the number of entities being monitored to those of current interest. A corollary is that for different entities of interest, it must be possible to use different ways of attaining scale, an example being the use of hierarchical monitoring structures like aggregation trees [28] for physical entities such as processors, blades, and racks vs. the use of peer-to-peer relationships for information dissemination and aggregation in distributed systems [21]. To permit such variety, we use *zones* as useful partitions of physical systems. Within each zone, data capture agents are connected to monitoring brokers using zone-specific structures, where the term ***monitoring topology*** refers to the structure used to connect data capture/analysis agents with data aggregators/monitoring brokers. Within the agents and brokers providing the physical containers for capturing and analyzing monitoring data, each specific monitoring task being performed is represented as a computational communication graph [3], and agents and brokers are internally multiplexed to operate any number of those graphs.

Runtime discovery, configuration, and adaptation – data center machines are subject to dynamic change in usage and load, but at the same time, we desire predictable latencies from monalytics. This means that it must be possible to change what, where, and how monitoring is done, including to deploy at runtime monitoring and analysis codes to where they are needed, to change the actual monalytics methods being used as well as the associated monalytics structures, and to dynamically discover and attach monalytics to new data sources when needed. Our work uses dynamic binary code generation and deployment to help obtain these capablities.

In summary, scalability demands that the monalytics components for data collection, aggregation, and analysis be flexible, ranging from simple centralized solutions to highly distributed ones. Further, solutions should scale up and down efficiently, particularly when monalytics drives real-time decision making like resource management to guarantee application SLAs and/or meet data center-level requirements like constraints on power usage [17].

Contributions and Results..

This paper makes the following technical contributions:

- The concept of monalytics is introduced, along with a software architecture for realizing it in large-scale data center systems.
- Scalability is also sought with respect to time and size scales, and to operate across the different levels of abstraction at which modern IT systems are described and implemented. The time scale is particularly relevant

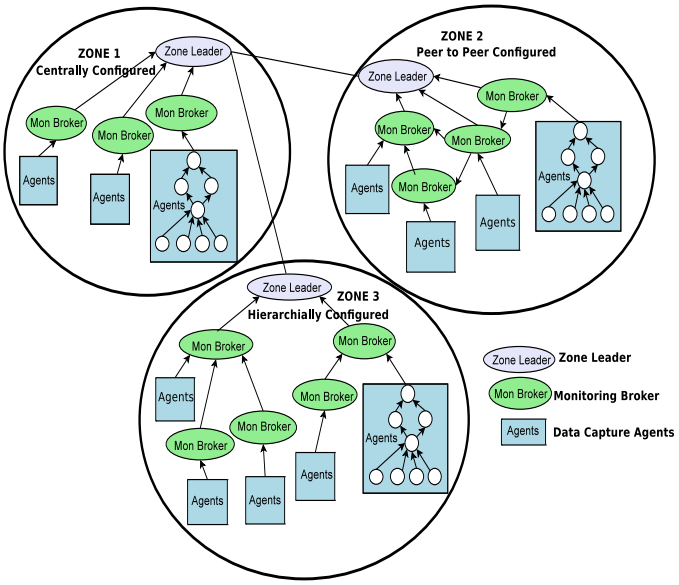


Figure 1: Monalytics Topology

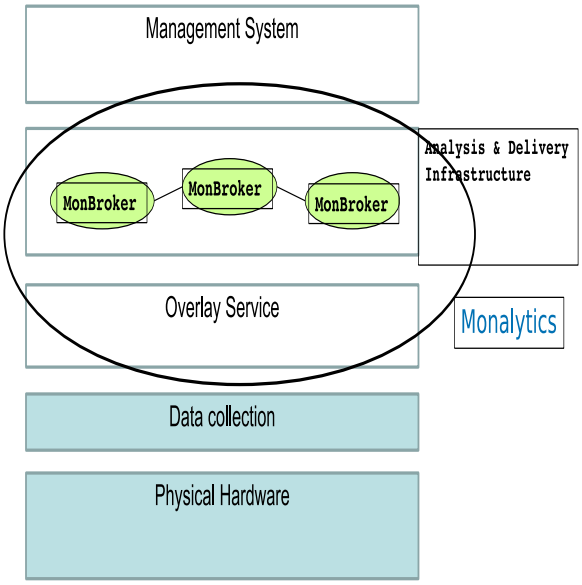


Figure 2: Monalytics Logical View

to online management, since it may be important to quickly react to dynamic changes in conditions and requirements.

- The utility of monalytics and its scalability principles are demonstrated with representative hardware and applications, and with micro-benchmarks.

Monalytics has been implemented for virtualized computing infrastructures. A prototype constructed for the Xen open source hypervisor is shown to add little to no additional overheads to the execution of typical data center codes. A transactional web application, variable request loads, and performance and fault behaviors are recognized and controlled, and the potential for scalability is demonstrated via adaptable filtering and 'failure-proportional' monalytics actions.

2. SYSTEM ARCHITECTURE AND IMPLEMENTATION

The realization of monalytics is based on three principles: (1) monalytics actions are deployed as a computational communication graph in the data center; (2) such graphs are elastic and reconfigurable based on monalytics needs, current state, and load in the data center; and (3) graph layouts and implementations are dynamic with respect to their use of centralized, hierarchical, or peer-to-peer structures, leveraging the best of the infrastructure but without having to subscribe to any one of them statically.

These principles give rise to the software architecture described in Fig. 1. *Agents* capture and locally process desired data; they reside at multiple levels of abstraction in target systems, including at application-, system-, and hypervisor-level. Agents also access hardware and physical sensors, such as hardware counters provided by computing platforms and power draw values exported by PDUs. *Brokers* aggregate and analyze outputs from multiple agents, and they are linked in ways that respect available communication and hardware structures. Each set of agents/broker are internally multiplexed to execute multiple logical structures – monalytics actions represented as computational communication graphs – that each represent specific captured data and the analysis methods applied to it. In other words, as with multiple threads in a single process, agents and brokers internally maintain and operate multiple monalytics graphs. Agent representations differ depending on target systems being monitored, including the use of specialized device drivers when interfacing agents with hardware data collection systems. Brokers can execute in specialized virtual machines (e.g., management VMs), on dedicated hosts (e.g., manageability engines), or both. Important in this context is that it must be possible to separately provision brokers and agents, so that latency and QoS guarantees can be made for monitoring and management, unaffected by current application actions and loads. Earlier work described how agents and brokers interact in a uniform manner, using a channel abstraction, termed m(anagement)channels [17]. The current implementation of monalytics has not yet been integrated with m-channels, but that work is in progress. *Zones* indicate physical subsystems, such as sets of racks on a single network switch in a data center, front end vs. back end machines, etc. Applications, typically comprised of *ensembles* of virtual machines, can span multiple zones, an example being web applications whose request schedulers run on data center front end machines, whereas its database servers run on the center's backend machines in a different zone. Zones, therefore, are a vehicle for delineating data center subsystems or substructures, or even the multiple physical data centers located in a single public/private cloud.

Applying these principles, Fig. 2 shows the logical view, with our core contribution being an overlay service and a collection of composed monitoring brokers that perform correlation, aggregation, and analysis functions. The overlay service provides the underlying communication/routing mechanisms as well as discovery and namespace registration for a collection of monitoring brokers. Together, they result in a monitoring computation graph (see Fig. 1), with multiple graphs multiplexed onto shared brokers, as also indicated.

The computational communication graphs used for monalytics perform the distributed monitoring functions in the data center in a cooperative manner. They are created and operate as described next.

The bootstrap creation phase starts by an initial assignment of monitoring brokers to logical zones by a planning algorithm at a central management station (CMS) [18]. Physically, a zone is a collection of nodes associated with a unique identifier communicated to the brokers on each physical node. After the assignment to zones, the monitoring brokers elect a leader for their respective zones using a leader election algorithm. The elected leaders of the respective zones then elect a leader among themselves (leader of leaders). The resulting inter-connection among the leaf node monitoring brokers, zone leaders, and inter-zone leaders is the initial monalytics graph after bootstrap. However, brokers only provide the physical containers that run monalytics graphs. This means that a primary role of leaders is to deploy and configure monalytics graphs across sets of brokers and supervise their execution.

During system operation, the role of monitoring brokers is to execute two functions: (1) they run the portions of monalytics graphs assigned to them, applying data aggregation and analysis functions on monitoring data streams, and raising alerts of anomalous behavior when detected (e.g., to enable local policies to be applied 'close' to the source of monitoring data); and (2) they propagate raw and/or analyzed local data streams to the other entities participating in each of their computation graphs, often ending at the zone-leader for zone-level aggregation. Along with such 'data plane' operations, there are 'control' actions, an example being a reaction to the fact that zone-level aggregation completion time exceeds an acceptable threshold. When that happens, the zone-leader may trigger reconfiguration of the monalytics graph within the zone, including those that switch the interconnections between monitoring brokers and leader from say, a centralized configuration to a multi-hierarchy topology, a completely peer-to-peer topology, or a combination of both.

When the aggregation for a polling interval completes, this triggers the execution of some management policy on the aggregated data. Processes like these occur across all levels of the monalytics graphs and may continue across zones, except that it now takes place among zone-level leaders. Similar reconfiguration of the computation graph (as within a zone) can take place across the leaders of the zones based on system load.

In summary, the monalytics system design leads to a proactive, elastic, and distributed monitoring system that supports hybrid topologies in a flexible manner. Its desirable properties include (1) applying analysis/anomaly detection policies local/close to source of monitoring data, (2) satisfying SLAs on aggregation/analysis completion times, such as an upper bound on those times, irrespective of scale in the system, (3) incurring overhead proportional and better than the scale and load in the system achieving elastic balancing, (4) flexibility of configuration and definition of SLA/metrics.

Fig. 3 depicts the Xen prototype of the monalytics infrastructure. The monitoring agents provide mechanisms to start, stop, and pause monitoring tasks. Each such task is represented as a computational communication graph, with graph nodes representing simple data processing operations and links denoting data transfers. Graphs can operate

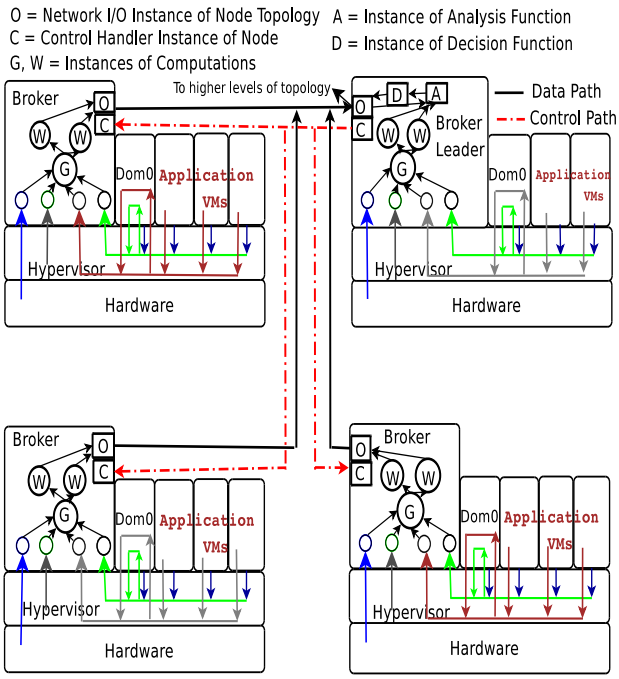


Figure 3: A Simplified Monalytics Architecture

within individual entities, such as an agent implementing a local feedback loop, and they can extend across multiple agents, brokers, and ultimately, zones.

The monalytics implementation separates control and data planes, where control actions can dynamically change the ways in which data is captured, processed, and transported. Current agents, for instance, are flexible with respect to how much and at what rates data is captured, and as to how data is combined and filtered. Most analyses are performed in brokers, including data aggregation from multiple agents. The current implementation maintains a list of predefined functions for these purposes, whereas runtime binary code generation and parametric controls are used to affect how agents operate. Selected brokers – leaders – run the 'coordinator' functions that control the data plane, i.e., create and re-configure the computational communication graphs that implement monalytics functionality.

We next describe in more detail the data capture agents and the analysis functions used in agents and brokers.

Data Measurement, Capture, and Analysis

The monalytics infrastructure offers several abstractions for capturing and analysis data:

1. ***Data capture*** – underlying monalytics are subsystem- and tool-specific components used for extracting data from systems and applications; to retain information about how this is done, monalytics maintains for each captured data item the tuple $\langle capture\_id, level\_id, sampling\_rate, sensor\_info \rangle$.
   $capture\_id$ represents the mechanism with which data is acquired, such as the unix */proc* or Xen's libvirt. $level\_id$ identifies the level of abstraction at which data is captured, such as the virtualization layer, operating system, platform or rack, etc. The actual data acquired in this fashion, sensor_info, is described as

$\langle time\_stamp, name, value \rangle$ tuple of metric data. The time_stamp represents the time at which metric is captured, and *value* contains the metric's value. $sampling\_rate$ represents the frequency of sampling monitoring metrics.

2. ***Data representation*** – once captured, data is represented as groups or windows. ***group*** – a group of data is always of some specific type and is captured in some context $group\_type, sensor\_info$, where measurement are collected in sets like cpu metrics, disk metrics, cpu and memory metrics e.t.c. and $sensor\_info$ is as above; an example is a group of cpu metrics that describe cpu utilization as $cpu\_system, cpu\_user, and\ cpu\_idle$ values; ***window*** – most analysis methods operate over window of values, motivating the implementation of a window as $\langle window\_type, window\_size, sensor\_info \rangle$ tuple; $window\_type$ can be of type time or event, $window\_size$ represents sizes such as 10sec or events, and $sensor\_info$ are as above.

To combine monitoring with analysis, monalytics permits low-level analysis actions to be associated with data capture. Such actions are represented as binary codes and can be customized to each specific data capture activity. Actions associated with data capture, for example, include a sample of violation counts for a group of metrics used – termed concise sample. Concise sample analysis is applied as post-processing actions to the event or time window actions to create a violation event for the last few seconds or events. It is likely that additional analysis actions will become built-in elements of data capture activities as the monalytics system evolves. Higher level analyses, however, are use- and application-specific; they are programmed explicitly and/or use mathematical and statistics libraries, with future work to offer interfaces to mathematical libraries like MatLab to assist end users in programming complex analysis actions.

Data Transport and Processing

The data transport and processing layer of monalytics uses the EVPath eventing system [23] to implement the computational communication graphs embedded in agents and brokers. Graphs are constructed as sets of linked *stones* traversed by *events* (i.e., structured data objects), where stones can perform event data filtering, data transformation, event multiplexing and demultiplexing, and event transmission to other stones. Stones can be linked within or across address spaces, the former via shared memory, the latter via standard communication protocols. Events are represented in efficient binary forms, via a portable binary format implemented in monalytics. Data is converted into this format upon entry to the monalytics system (e.g., via XML-to-binary conversion). Each stone is associated with a set of *actions*. Actions are codes that operate on the messages/events handled by the stones. Stones also have the ability to temporarily hold events, retain limited state, and can be created or destroyed at runtime. Finally, higher level actions defined on sets of stones can reconfigure them and their linkages [18].

Stones can carry out arbitrary actions. (a) An *output action* causes a stone to send messages to a target stone across a network link. (b) A *terminal action* specifies an application handler that will consume incoming events. (c) A *filter action* allows handlers that filter incoming data to determine if it will be passed to subsequent stones. (d) A *split action* allows incoming events to be sent to multiple stones. The target stones of a split action can be dynamically changed by adding/removing stones from the split target on the fly. (e) A *transform action* transforms event data from one data type to another, and it can be used to perform complex calculations on events, such as sampling, averaging, and compression. Actions are implemented by handlers written in C or via runtime generated binary codes, the latter specified in E-Code, a portable subset of the C language. Runtime action deployment, then, uses dynamic linking or dynamic code generation, respectively.

Monalytics implementations can be built on substrates other than EVPath, of course, but there are multiple principles embedded in EVPath that contribute to the scalable design of monalytics. These include the runtime creation and deployment of meaningful data capture and analysis actions, the ability to dynamically reconfigure monalytics (both in terms of actions and graph structure), the capability to operate efficiently both 'in the small', e.g., in a single address space, and 'in the large', i.e., across the many agents and brokers present in a large scale data center. Efficiency 'in the small' is provided in part by use of compact binary representations of data coupled with compiled action implementations. The compact binary formats in current use include formats that define groups, event and time windows, encodings for various metrics, etc. Efficiency 'in the large' is supported by runtime reconfiguration of the monalytics structures spanning many agents and brokers.

The default EVPath execution model is push-based, where agents continuously capture sensor data, process it via local actions, and send it to brokers (as defined by monitoring graphs), and brokers carry out global actions. To support adhoc monalytics queries, we have added a pull-based model, where queries posed to brokers cause agents to acquire and provide certain data, as and when needed. In addition, we are currently implementing efficient techniques for storing or caching select and/or adhoc state present in monalytics structures, a concrete example being a DHT used to maintain state for popular or common queries [26]. The methods available for controlling how monalytics graphs operate are described next.

Controlling Monalytics

Monalytics is implemented to separate control from data paths, where data capture, processing, and forwarding are done via monalytics structures, and control actions are taken by coordinators associated with these structures. Technically, this means that agents and brokers jointly execute monalytics graphs, but that select brokers like zone leaders also run coordinator processes that control how monalytics graphs function.

Control actions of interest to this paper include the ability to dynamically change (a) the capture rate for monitoring metrics data, (b) the monitoring graphs' processing points, i.e., actions, (c) the targets for the processing points in the monitoring graphs and between agents and brokers, the latter enabling higher level functions that reconfigure graph topologies. For (c), for instance, a *split* stone can use a 'target list' of various targets to be used, and this list can be changed at runtime. Changes may be triggered by stone arrival/departure or by link creation/deletion. The experimental evaluations in Section 4 use (a) and (b), for instance, when dynamically switching from lightweight monitoring for

abnormal behavior detection to undertaking diagnostic monitoring for problem resolution.

Implementation Comments

The current monalytics prototype is implemented in C/C++ using EVPath as its base communication layer. Our next implementation steps concern higher level facilities like those needed for leader election and like runtime structure reconfiguration (e.g., see our earlier work on iFlow [18] for ways to implement such functionality). We have not yet integrated monalytics with the vManage infrastructure described in [17], a first step in that work now underway being the placement of brokers into separate 'Management VMs' (virtual machines dedicated to carrying out monalytics functions) that can be deployed to machines at runtime when and if needed. This means that currently, the agents placed into Xen's dom0 (the 'driver domain') use statically created links to pre-created Management VMs containing brokers. Finally, the data capture methods used in monalytics exploit hypervisor level facilities like libvirt, system-level APIs like /proc, and select adhoc application-level monitoring. For generality, future work will deal with the use of monitoring standards like CIM, additional capture methods like those exploiting xentrace, and interfacing to higher level support for application-level monitoring provided by systems like Ganglia [19] or Tau [22].

3. THE NEED FOR MONALYTICS

The development and evaluation of monalytics principles, functionality, and infrastructure are driven by typical data center use cases. These cases and their motivation are described next. Additional detail appears in Section 4 of this paper.

Understanding Application and System Behaviors..

For large-scale data centers, it is critical to understand the dynamic behavior of infrastructure and systems, as well as of the applications running on data center hardware. An operator, for instance, will want to ascertain 'current health' by asking high level queries about certain system or machine states. Such queries go beyond simple questions about which machines currently appear to be up or down to questions that include the following: (1) are certain machine and subsystem (e.g., disk or network) utilizations within certain bounds, or (2) what is the current utilization of processors, memory, disk, network subsystems, or (3) what are the thermal and power draw values normalized by current system utilization, or (4) are certain application SLAs being met?

As evident from these questions, the data required for answering them must be captured at and traverse multiple levels of abstraction (e.g., platform sensors used for thermal or power draws vs. application-level measurements for ascertaining SLAs), and even conceptually simple questions like whether or not SLAs are being met will typically require captured data to be analyzed over certain time windows and for certain statistical likelihoods, perhaps even including economic or risk analyses [8]. Also evident is the fact that there is considerable parallelism in how such questions are answered, an example being the many entirely concurrent platform-local analysis actions that aggregate utilization values in conjunction with power draw and thermal data. At the same time, there is a necessity for global actions like aggregation in order to display results to operators and more importantly, to understand cross-cutting behaviors, perform historical analyses, and enter data in long term logs. Finally, the sources of questions like the above are not just human operators, but also automation engines like those performing datacenter provisioning, accounting, compliance checking, intrusion monitoring or spam detection, or similar continuous management tasks.

Questions like those above motivate several functionalities of monalytics. (1) Data-local actions are important to encourage parallelism in analysis. (2) Data reductions are enabled by combining data capture with analysis, as when reporting window averages rather than entire windows (e.g., to answer questions about current utilization) or when reporting bound violations for certain systems or applications. As stated earlier, both (1) and (2) are critical to attaining the scalability across time and length scales and levels of abstraction required for large-scale system management. (3) A final functionality is the need for dynamic control over what and how monitoring is performed, in conjunction with control over where decisions are made concerning the reactions to discovering certain phenomena: locally, globally, or both. An example is load balancing on a single multi-core platform to vacate as many cores as possible to reduce platform power draw, driven by temporary variations in application behavior and concurrency, coupled with additional, lower rate virtual machine movement across many platforms and racks for consolidation and power savings purposes, the latter perhaps driven by diurnal changes in load.

Experimental evaluations in Section 4 demonstrate the basic functionality of monalytics and establish the benefits of using source-near filtering or analysis. We also demonstrate our ability to dynamically control how and what monitoring is carried out.

Assessing and Managing Data Center Systems..

A typical issue in utility data center and cloud computing systems is misbehavior triggered by overloads, failures [4], or unusual application or system conditions. Such problems must typically be recognized and addressed quickly [9], in order to prevent congestion from worsening, failures from spreading to other subsystems, or the occurrence of catastrophic events like shutdowns. Continually assessing applications and systems, the paper's experimentation section evaluates the ability of monalytics to deal with a concrete scenario concerning failures. Three facts are of importance to monalytics in this context: (1) the ability to quickly recognize issues, by continuously and with low overhead checking relevant system or application behaviors on all nodes and for all application components involved; (2) the ability to link runtime detection to corrective actions, where 'local' linkages are key to reacting with low delay; and (3) the capability of dynamically moving from lightweight supervisory monitoring to detailed monitoring for problem diagnosis. Further, such actions must again be taken across multiple levels of abstractions, depending on the ability and methods used to detect overloads or failures.

Meeting Application Requirements..

Monalytics can be used to adjust and manage systems and applications in order to provide certain guarantees to end users, including response times, throughput levels, etc. It is typically not possible to implement such functionality with-

out operating across the multiple levels of abstraction and many subsystems existing in IT infrastructures. Further, since applications can be mapped onto different machines, including dynamically through runtime consolidation, monalytics must be capable of 'following' application components to wherever they run. Finally, recognizing the reasons why application requirements are not met can be quite complex. It may require detecting whether a system has failed or not, for example, and it typically requires global in addition to local analysis actions. Global analysis in turn requires monalytics to use aggregation trees and similar data dissemination structures to make appropriate data available to sophisticated methods for cause recognition and decision making.

The next section uses concrete instances of the general use cases described above to demonstrate and evaluate associated monalytics functionality.

4. EXPERIMENTAL EVALUATION

The experimental evaluation is divided into multiple subsections. The first subsection, describes the testbed, workload and target system used. The second subsection describes specific experimental scenarios and their use of monalytics features with results.

The experiments used in this section use a small testbed comprised of several multicore servers running the RUBiS [10] three-tier web services benchmark's servlet version on virtualized environment using the Xen hypervisor, with Apache, Tomcat, and MySQL each using different domUs. For some experiments we also use an open loop workload generator [24] and a simple load balancer [1]. Monitoring agents code run on each of the backend nodes. The agents, by default, monitor the cpu and memory utilization metrics of domUs and dom0.

Experimental Scenarios

Monalytics is evaluated in specific scenarios representing typical datacenter events or behaviors. The first two scenarios model misbehaviors due to software bugs and application misconfiguration, leading to corrective actions performed at different levels of abstraction. The third scenario demonstrates scalability achieved through data local analysis, using monalytics filtering functionality. Finally, we show how monalytics can be used in a large-scale decision analysis method [27]. In each of these scenarios, the cpu and memory utilization with monalytics infrastructure is less than 2% on brokers and broker leaders.

Runtime Component Misbehavior

This scenario demonstrates the usefulness of monalytics to 'assess and manage' data center systems. Here, we dynamically instantiate a monalytics control loop to deploy a corrective action, when monitoring and fault detection identify a faulty backend RUBiS instance.

We recreate an apache bug that causes segfaults and finally stops all interactions between certain RUBiS components. The effect of the bug is also observed at the loadbalancer's receive channel that interacts with the respective RUBiS instance. In this experiment, the behavior is detected by monitoring (1) the cumulative packets transmitted between apache, tomcat, and mysql domUs, and (2) the cpu utilization of RUBiS VMs along with the cumulative number of bytes received via the loadbalancer channels during the experiment run. Misbehavior is diagnosed when each of these metrics show no change in their values over some period of time.

Table 1: Impact Of Monalytics On End User Metrics

Case	Total Requests	Unsuccessful Requests
W/O Control Action	53535	13976
With Monalytics	52535	5763

For experimental runs, we inject one of the five RUBiS instances with the buggy Apache VM, generate workload requests at 1000requests/sec for 600sec, and start monitoring at 5 sec sampling rate.

With monalytics, fault detection is followed by a decision operation that sends an event to a trigger operator to trigger a corrective action, thus instantiating the aforementioned control loop. The action used simply restarts the Tomcat and Apache servers of the faulty RUBiS instance. Toward this end, we export libvirt's actuation APIs to start/stop a VM and then use the runtime code generation facilities of monalytics that calls these APIs.

Table 1 demonstrates the utility of associating simple analysis actions directly with data capture, as supported by monalytics. Utility is measured in terms of end user metrics – total unsuccessful requests over the period of the experiment run. During the run, once a fault is detected, analysis action recognize the application's need for corrective action, to prevent the Apache bug from unduly damaging application progress. This action is realized in the monalytics infrastructure, i.e., it is created via dynamic binary code generation and deployed on the faulty node. Similar in spirit to microreboots [9], it simply stops/starts the VM in question.

Table 1 reports the utility measured over the period of experiment run in two cases: (a) in worst case, if we do not take any corrective action and (b) using the monalytics-deployed corrective action. As known from prior work [9], such corrections help reduce the number of unsuccessful requests, in this case from 26% to 11%. More generally, the example demonstrates two important elements of the monalytics approach to system management: (1) runtime code generation and deployment of corrective actions, since one cannot assume prior knowledge of all actions that might be used in a large-scale system, and (2) the use of local vs. global control, that is, the placement of control actions 'near' where issues occur, to prevent undue monitoring traffic and encourage small delays between failure detection and reaction.

Performance Aware Load Balancing

This scenario demonstrates the use of continuous monalytics for to help applications meet their requirements (i.e., SLAs).

We emulate a degradation in the performance of critical bidding requests on one of the RUBiS instances servicing them. This is done by manipulating the maximum number of workers (MaxClients) and server limit (Server Limit) parameters on the Apache VM used by this instance. Manipulations cause longer processing time for bidding requests on this vs. other RUBiS instances, the effects of which are also observed at the loadbalancer's receive channel interacting with the instance. In the experiment, this behavior is detected by monitoring (1) the current number of busy workers and the request processing rate in each Apache VM and

Table 2: Impact Of Monalytics On End User Metrics

Case	Total Requests	Requests Meeting SLA
W/O Control Action	32213	21334
With Monalytics	32213	27763

(2) the cumulative number of bytes received via the loadbalancer channels during the experiment run along with the cpu utilizations of RUBiS VMs. Performance is considered degraded when these metrics consistently exhibit lower values for one RUBiS instance compared to others over some period of time.

There may be many causes for misbehavior, of course, but the objective of this experiment is not to identify failure causes but instead, to demonstrate the need for certain monalytics functionality. Specifically, monalytics must not only be able to to change how monitoring and analysis are done, but should also be able to dynamically inject simple management or corrective actions at different levels of abstraction and/or in different subsystems. In the previous experiment, corrective actions were taken by the hypervisor in monalytics leaf nodes, via VM stops/starts. In this experiment, actions are application-specific and are associated with the loadbalancer, i.e., in non-leaf nodes of monalytics topologies.

For these experiment runs, we inject misconfiguration on one of the three RUBiS instances serving bidding requests, generating workload at 1500requests/sec for 600sec. As the experiment proceeds, misbehavior detection causes the instantiation of a corrective action in the loadbalancer. The action simply flags the loadbalancer's degraded channel and then uses this information to change future load balancing decisions. This is implemented by modifying and exporting the loadbalancer API to mark up loadbalancing decisions and then again using runtime code generation to create codes on the loadbalancer node that call these APIs. We observe that such functionality can be implemented and changed separately from systems and application codes.

Table 2 reports the utility measured over the period of experiment run in two cases: (a) in the worst case, if we do not take any corrective action and (b) using monalytics to deploy a corrective action. Online monitoring and analysis via monalytics, coupled with the runtime installation of a simple control action, leads to an increase from 66% to 86% in the number of bidding requests that meet deadlines.

Scalability via Local Analysis

One way to attain scalability in monitoring is to immediately analyze data to the extent possible. A known useful purpose of local analysis is to use it to filter monitoring information, thereby reducing total data volumes passed across monalytics topologies. Runtime code generation and deployment can be used to attain this goal, as demonstrated next.

Data filtering is particularly important for high volume monitoring tasks like tracing. As an example, we trace the http requests processed by the Apache webserver. The trace record includes several fields, including: server_timestamp, request_url, request_params, request_time, client_ip, server_ip. In the experiment, we inject requests at 100 requests/sec for 600sec on a single webserver, and trace data is sent to the broker leader. The leader tracks the requests and their processing times and once a defined number of requests crosses some processing time threshold (200 ms), this triggers the decision to deploy a 'filter' operator at the apache webserver. This operator analyzes request trace data to send to the broker leader only those requests that have processing times above 200 ms and for such requests, it only forwards their url and running counts.

Without runtime filtering, tracing quickly leads to large monalytics volumes and overheads, resulting in 1.46 MB of request trace records for a 600 sec run. With filtering, the broker leader receives only 60.45KB of filtered data. This is because as with many such uses of tracing, in this experiment, only 2.3% of all requests exceed the stated processing time threshold, resulting in much data being transferred 'uselessly' when filtering is not used. More importantly, in actual systems, filtering criteria are dynamic, depending on current conditions and requirements.

Zoom-In Analysis

Building on the previous example demonstrating the importance of runtime data filtering, we next describe a more realistic set of decision methods used for this purpose. The goal is to demonstrate potential scalability via ***'failure-proportional'*** rather than *'system size-dependent'* monalytics actions.

The methods used here are based on our ongoing development of the EBaT lightweight, anomaly detection methods [27]. We again monitor the RUBiS application, where an agent in each machine's dom0 collects local virtual machine metrics (vcpu utilizations) and calculates a local entropy timeseries. These metrics, entropy timeseries, are then sent to a broker. Local entropy analysis results in low messaging overhead, because the metric transferred is a single entropy value of type float. The broker collects values provided by all agents and computes a global entropy timeseries. This ensures that any anomalies observed in the local entropy timeseries are reflected in the global timeseries, as well. A global entropy decomposition process is triggered after an anomaly in the global timeseries is detected. As the coordinator has the composite of each global entropy value, it can then 'zoom in' to the appropriate local entropy value/values that contribute to the abnormal global value change, thereby identifying the associated servers. Upon identification, additional monitoring and analysis are triggered in the appropriate local agents to further diagnose the anomaly, with sample values captured including application level data like number of busy threads, request rates, etc. This is done using a decision tree constructed using the machine learning method for system monitoring and failure diagnosis proposed in [12].

With zoom-in, rather than always monitoring all possible values of interest for detailed problem identification, monalytics' runtime methods for code deployment are used to dynamically install detailed monitoring only on where and when it is required. This means that monalytics overheads are proportional to the severity of failures – failure-proportio-nality – rather than being dependent on raw system or application size. To illustrate, consider the use of zoom-in analysis in large scale data centers, with a representative system considered in our work with EBaT [27] comprised of 81,920,000 virtual machines. Conservative estimates using say, one virtual machine to deploy one Apache server and observing 50 different monitoring metrics, results

in over 4G of monitoring data (assuming 10 bytes per metric per second) and in 40G bytes of data per second transferred to logging servers, generating undue data volumes and overheads. In contrast, when using the dynamic code generation and deployment capabilities of monalytics, runtime filtering operators use custom local decision trees for data-local analysis, moving select computations to data rather than moving excessive data amounts to analysis or logging nodes.

To demonstrate the feasibility of zoom-in analysis, a 10 hour experiment is measured for 3 hours, with 100 anomalies randomly injected into the RUBiS application. We compare a centralized method that (1) gathers all of the metrics from all agents, and (2) uses a typical threshold-based method for anomaly detection [27] with the proposed EBaT method. As shown in [27], EbAT outperforms the common threshold-based approach in terms of anomaly detection rate, false alarm rate, and accuracy. Similar results are obtained with the experiment run for this paper, but those are not shown for reasons of brevity. More important to this paper is the fact that with the zoom-in method, the monalytics overlay at runtime transfers a total of only 123.32 KB of local decision data, whereas almost three times as much data, 394.08KB, is transferred with the centralized solution. With offline analysis, for 10 hour runs, the centralized solution generates 1.15 MB of data against a 345.6KB data volume with data-local analysis. This again indicates that scalability can be attained by leveraging the dynamic and data-local processing capabilities of monalytics. Simple qualitative arguments for larger monalytics topologies can be used to further underline this argument.

5. RELATED RESEARCH

There are many partial or subsystem-level solutions to systems management. At one end of the spectrum, there are rich all-encompasing commercial monitoring and management solutions such as HP System's Insight Manager, IBM's Tivoli, and VMware's vCenter for data center environments. These systems perform centralized data collection and analysis, and provide some support for script-based triggering mechanisms. There is also hardware-level support focused on certain physical subsystems, such as HP's iLO or IBM's Director solutions for blade centers. None of the these solutions currently scale to the sizes needed in next generation data center systems.

There exist several open source tools for collecting monitoring data and for cluster-level monitoring [19]. [19] uses a hierarchical approach to monitoring where attributes are replicated within clusters using multicast methods and aggregated via a tree structure. Aggregation structures are evaluated in several related efforts, including [21,26,28]. [15] supports on-demand but not complex queries. [6] constructed on top of hadoop provides monitoring and analysis for large data-intensive codes and systems, focused on large volumes logs for failure diagnosis.

Note that most of the projects described above focus on scalability in data distribution and aggregation, supporting continuous or one-shot queries via certain hierarchical or peer-to-peer topologies, and they may use gossiping techniques or data structures like DHTs to access and distribute monitoring data. The monalytics approach to large-scale data center management can leverage the robustness and scale properties of such methods, but differs in also (1) combining data collection and aggregation with arbitrary analysis tasks, (2) permitting dynamic deployment and reconfiguration of monalytics graphs and operators, (3) providing scalability through data local analysis, and finally, (4) extending analysis with local management functions.

Monalytics leverages some of the concepts from earlier work on data streaming systems, including [5, 25], and our own research on the ECho publish/subscribe system [13] and the iFlow [18] event-based infrastructure, applied to high performance and to enterprise scale systems.

6. CONCLUSIONS AND ONGOING WORK

This paper presents the concept of monalytics along with the software architecture for realizing it in large scale data center environments. It provides concrete data center usage scenarios to establish the need for monalytics. Experimental evaluations demonstrate how simple actions integrated with monitoring can be helpful, particularly when it comes to issues of scale. We demonstrate the usefulness of monalytics through its features – deploying local control loops as corrective actions, installing them at different levels of abstraction or in different subsystems via monalytics leaf- or non-leaf nodes, and attaining scalability through data local analysis and failure proportionality for larger scale systems or tasks.

Ongoing work in this project includes (1) the integration of monalytics with the vManage [17] architecture, (2) its deployment in larger scale testbeds, including those provided by the OpenCirrus cloud computing infrastructure [2], and (3) additional experimental evaluations ranging from micro-benchmarks, to using parallel analysis operators, to demonstrating multiple distributed control loops in larger scale applications operating across different length and time scales.Also in progress are larger-scale experiments on the high performance machines accessible to our group [3].

Future work with monalytics concerns our ability to use and manage monalytics topologies, including dynamic topology configuration and the use of additional data aggregation and organizational methods, such as DHTs [26]. We are also exploring power-performance tradeoffs [17] in utility cloud computing systems, through detailed measurement and evaluation of commodity servers in an instrumented data center at Georgia Tech, including consideration of thermal and cooling issues.

7. REFERENCES

[1] Balance http://sourceforge.net/projects/balance/.

[2] Open Cirrus HP/Intel/Yahoo Open Cloud Computing Research Testbed https://opencirrus.org/.

[3] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. *HPDC*, 2009.

[4] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham. E2EProf: Automated End-to-End Performance Management for Enterprise Systems. *DSN*, 2007.

[5] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. *ICDCS*, 2006.

[6] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa: A Large Scale Monitoring System. *Cloud Computing And Its Applications*, 2008.

[7] F. Bustamente, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active Streams: An Approach to Adaptive Distributed Systems. *HotOS-VIII*, May 2001.

[8] Z. Cai, Y. Chen, V. Kumar, D. S. Milojicic, and K. Schwan. Automated Availability Management Driven by Business Policies. *IM*, 2007.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - A Technique for Cheap Recovery. *OSDI*, 2004.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. *OOPSLA*, Nov 2002.

[11] H. Chen, G. Jiang, K. Yoshihira, and A. Saxena. Ranking the Importance of Alerts for Problem Determination in Large Computer Systems,. *ICAC*, 2009.

[12] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure Diagnosis Using Decision Trees. *ICAC*, 2004.

[13] G. Eisenhauer, F. Bustamente, and K. Schwan. Event Services for High Performance Computing. *HPDC*, 2000.

[14] D. Gupta, R. Gardner, and L. Cherkasova. XenMon: QoS Monitoring and Performance Profiling Tool http://www.hpl.hp.com/techreports/2005/HPL-2005-187.html.

[15] J.Liang, S.Y.Ko, I.Gupta, and K.Nahrstedt. MON: On-Demand Overlays For Distributed Systems Management. *Workshop on Real, Large Distributed Sytems*, 2005.

[16] E. Kiciman and B. Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. *SOSP*, 2007.

[17] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. *ICAC*, 2009.

[18] V. Kumar, Z. Cai, B. F. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing Diverse Messaging Models with Self-Managing Properties using IFLOW. *ICAC*, 2006.

[19] M.L.Massie, B.N.Chun, and D.E.Culler. The Ganglia Distributed Monitoring System: Design, Implementation and Experience. *Parallel Computing*, 2004.

[20] J. H. Perkins, S. Kim, S. Larsen, et al. Automatically Patching Errors in Deployed Software. *SOSP*, 2009.

[21] R.V.Renesse, K.P.Birman, and W.Vogels. Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management and Data Mining. *ACM Transactions on Computer Systems*, 2003.

[22] T. Sheehan, A. Malony, and S. Shende. A Runtime Monitoring Framework for the TAU Profiling System. *International Symposium on Computing in Object-Oriented Parallel Environments*, December 1999.

[23] M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, 1992.

[24] C. Stewart, T. Kelly, and A. Zhang. Exploiting Nonstationarity For Performance Prediction. *Eurosys*, 2007.

[25] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. *International Symposium on Software Reliability Engineering*, 1998.

[26] S.Y.Ko, P. Yalagandula, I.Gupta, V.Talwar, D.Milojicic, and S.Iyer. Moara: Flexible and Scalable Group Based Querying Systems. *Middleware*, 2008.

[27] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online Detection of Utility Cloud Anomalies Using Metric Distributions To appear. *NOMS*, 2010.

[28] P. Yalagandula and M. Dahlin. SDIMS: A Scalable Distributed Information Management System. *SIGCOMM*, 2004.

The Spineless Tagless G-machine, naturally

Jon Mountjoy

Department of Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
email: jon@wins.uva.nl

Abstract

The application of natural semantic specifications of lazy evaluation to areas such as usage analysis, formal profiling and abstract machine construction has shown it to be a useful formalism. This paper introduces several variants and extensions of this specification.

The first variant is derived from observations of the Spineless Tagless G-machine (STG), used in the Glasgow Haskell compiler. We present a modified natural semantic specification which can be formally manipulated to derive an STG-like machine.

The second variant is the development of a natural semantic specification which allows functions to be applied to more than one argument at once. The STG and TIM abstract machines both allow this kind of behaviour, and we illustrate a use of this semantics by again modifying this semantics following observations of the STG machine. The resulting semantics can be used to formally derive the STG machine. This effectively proves the STG machine correct with respect to Launchbury's semantics.

En route, we also show that update markers in the STG machine are necessary for termination, and show how well-known abstract machine instructions, such as the *squeeze* operation, appear quite naturally as optimisations of the derived abstract machine.

1 Introduction

The STG machine has proved itself as being capable of efficiently executing lazy functional languages. This machine lies at the heart of the Glasgow Haskell Compiler (Peyton Jones 1996), which compiles by translating a program written in Haskell through intermediate, simpler, languages until it finally generates programs in the STG language. This final language is then compiled into code which mimics the operational semantics given to the language in the form of an abstract machine. However, it is yet to be shown that the STG abstract machine is correct. Intuitively the machine does what we expect it to do, being a refined model of a graph reducer which, operationally, seems sound.

The first natural semantic specification of lazy evaluation for lazy functional languages was presented in Launchbury (1993), which was proved correct with respect to a call-by-name denotational semantics of the same language. We regard this as *the* specification of what is meant by a lazy functional language. Sestoft (1997) has taken the natural semantic specification and derived from it various abstract machines, proving that these abstract machines are correct with respect to each other and the natural semantic specification. It is not quite clear, however, how these abstract machines can be related to the STG machine.

This paper introduces several variants of natural semantics specifications. The first variant is derived from observations of the STG, which seems to treat variables pointing to lambda abstractions as values, as opposed to just lambda abstractions. A modified natural semantic specification is suggested which does just this, and we show how it can be formally manipulated to derive an STG-like abstract machine.

The STG machine, however, just like the TIM abstract machine, is capable of handling multiple arguments in its applications and abstractions. Unfortunately, Launchbury's semantics handles only single arguments. Our second variation extends Launchbury's semantics to multiple arguments. The STG machine can then be formally derived from this semantics, yielding a proof that the STG machine is correct.

En route, we also show that update markers in the STG machine are necessary for termination, and show how well-known abstract machine instructions, such as the *squeeze operation, appear quite naturally as optimisations of the* derived abstract machine.

The structure of this paper is as follows:

- The following section recalls the natural semantics of lazy evaluation which we will be using as a reference semantics.
- Section 3 makes observations about the STG which suggest changes to the natural semantic specification. A new semantics is then developed and shown correct with respect to the reference semantics.
- Section 4 illustrates how an STG-like abstract machine can be derived from the semantics.
- Section 5 extends the reference semantics to handle multiple arguments during abstraction and application.
- Section 6 parallels sections 3 and 4 and derives a semantics for the STG machine based on multiple arguments, and subsequently the STG machine.

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad Lam$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : w}{\Gamma : e\ x \Downarrow \Theta : w} \qquad App$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto w] : \hat{w}} \qquad Var$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \Downarrow \Delta : w}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \Downarrow \Delta : w} \qquad Let$$

$$\Gamma : c\ x_1 \ldots x_j \Downarrow \Gamma : c\ x_1 \ldots x_j \qquad Cons$$

$$\frac{\Gamma : e \Downarrow \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \Downarrow \Theta : w}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n \Downarrow \Theta : w} \qquad Case$$

Figure 1: Natural Semantic specification of Lazy Evaluation

Unfortunately the STG machine is quite large and complex, and to describe it here would require too much space. We thus assume some knowledge of the STG machine, as presented in (Peyton Jones 1992). Proofs of all of the propositions will be made available in a technical report.

2 The Natural Semantics of Lazy Evaluation

We begin by (briefly) reviewing the language and semantics based on Launchbury (1993).

The abstract syntax of the language, which we christen the *basic* language, is given by:

$$\begin{array}{lrcl} \text{expressions:} & e & ::= & \lambda x.e \\ & & | & \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \\ & & | & e\ x \\ & & | & x \\ & & | & c\ x_1 \ldots x_j \\ & & | & \text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n \end{array}$$

This syntax guarantees that expressions are in a so-called *normalised* form. The process of normalisation ensures that:

- application of a function can only be made to a single variable
- all constructors (c) have only variables as arguments
- all constructors are saturated (fully applied)
- all bound variables are *distinct*

The first three properties are ensured by the syntax. The last is implemented by an α-conversion which renames all bound variables using completely fresh variables, which we write as $\hat{e}$. Launchbury (1993) provides a simple scheme for normalising arbitrary expressions into the basic language. Intuitively, the restriction of application and constructor arguments to variables makes the sharing of these arguments explicit. Note that Launchbury only allows application and abstraction of a single argument. A natural semantic specification of this language is shown in Figure 1.

In the above, a heap, $\Gamma = [\ldots, x \mapsto e, \ldots]$ represents a mapping from variables x to expressions e. We write $\Gamma[x_i \mapsto e_i]$ for $\Gamma[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$, *dom* Γ for the domain of Γ, and *rng* Γ for the range (the set of expressions bound in the heap) of Γ. The list of alternatives of a case statement is written as *alts*(below) or $\{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n$, where $\vec{y_j}$ represents the vector $y_{j_1} \ldots y_{j_{a_j}}$ of variables, a_j being the arity of the constructor c_j. A *configuration* $\Gamma : e$ is an expression and a heap in which the expression is to be evaluated. A *judgement* $\Gamma : e \Downarrow \Delta : w$ says that expression e in heap Γ evaluates to a value w and heap Δ.

In Figure 1 we see that rules *Lam* and *Cons* say that the value of the lambda abstraction or constructor is itself – indeed, these are what lazy functional languages consider to be values!

Rule *Let* evaluates a let expression by evaluating the qualified expression e in a heap in which all of the bindings are present. Note that this is the only rule which adds bindings to the heap, and that the bindings themselves are not evaluated.

Rule *App* says that to evaluate an application $e\ x$, we need evaluate e to a lambda abstraction $\lambda y.e'$, and evaluate the e' with x substituted for y in the resulting heap to produce a final value and heap.

An integral part of lazy evaluation is that when something is evaluated, the result is rebound in the heap so that further requests for the value yield the already evaluated value. Rule *Var* captures precisely this. This rule says that to evaluate a variable bound to an expression e in the heap, we can evaluate this expression in a heap without this binding to produce a value. This value can then be *rebound* to the variable in the final heap – thus ensuring that further demands of the variable will yield the new value. This is the only place where a heap *update* occurs. The STG language of section 3 provides support for sometimes avoiding this expensive operation, if it will lead to no change in the final value produced. Since we duplicate value w, this may cause name clashes (recall that otherwise all bound variables are distinct) and we have to rename all of the bound variables of w to fresh variables, indicated by $\hat{w}$. See Launchbury

$$\Gamma[x \mapsto \lambda y.e] : x \;\downarrow\; \Gamma[x \mapsto \lambda y.e] : x \qquad Lam_S$$

$$\frac{\Gamma : f \;\downarrow\; \Theta[v \mapsto \lambda y.e] : v \quad \Theta[v \mapsto \lambda y.e] : e[x/y] \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : f\ x \;\downarrow\; \Delta[u \mapsto w] : u} \qquad App_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[u \mapsto w, x \mapsto \hat{w}] : x}\ w \text{ a lambda abstraction, } e \text{ not} \qquad Var_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c\ y_1 \ldots y_j}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[x \mapsto c\ y_1 \ldots y_j] : c\ y_1 \ldots y_j}\ e \text{ not a lambda abstraction} \qquad VarC_S$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \;\downarrow\; \Delta[u \mapsto w] : u} \qquad Let_S$$

$$\Gamma : c\ x_1 \ldots x_j \;\downarrow\; \Gamma : c\ x_1 \ldots x_j \qquad Cons_S$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \;\downarrow\; \Theta[u \mapsto w] : u}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n \;\downarrow\; \Theta[u \mapsto w] : u} \qquad Case_S$$

Figure 2: Natural Semantic specification of Lazy Evaluation for the λ-normalised language

(1993) and Sestoft (1997) for more on this operation and its relation to name clashes.
Rule *Case* evaluates an expression to a constructor, and then selects the appropriate expression from the list of alternatives, substituting the arguments of the constructor for the formal variables in the alternative.

Launchbury proves that the natural semantic specification is sound and complete with respect to a denotational semantics of the language:

Proposition 1 (Denotational Correctness for $\Downarrow$)
If $\Gamma : e \;\Downarrow\; \Delta : z$ *then for all environments* ρ *we have:* $[\![e]\!]\rho\Gamma = [\![z]\!]\rho\Delta$

Sestoft (1997) has shown how an abstract machines can be formally derived from this natural semantics specification. He thus derives an abstract machine $\mathcal{A}$ which simulates $\Downarrow$ in the sense that the abstraction machine will yield a value for an expression iff the natural semantics proves that the value of the expression e is w. Writing $\xrightarrow{\mathcal{A}}$ for a step in the abstract machine $\mathcal{A}$, and $\xrightarrow{\mathcal{A}}{}^*$ for its transitive closure, we have:

Proposition 2 (Correctness of Sestoft's Machine)
If w *is a value then* $e \;\Downarrow\; w$ *iff* $e \xrightarrow{\mathcal{A}}{}^* w$.

If we can use these techniques to formally derive the STG machine, then we will have provided a proof of correctness of the STG machine, and related it to Sestoft's machine. Unfortunately, it is not quite clear how this should be done. The following two sections show a way – we first highlight characteristic features of the STG machine and modify our semantics to incorporate them, and we then extend the entire semantic framework to handle multiple arguments. As a result, we produce a semantics from which the STG machine can be derived.

3 The STG Machine

The syntax of the STG machine base language, as presented in Peyton Jones (1992), is quite different from that of the basic language considered in the previous section. These syntactic differences contribute significantly to the differences in the derived abstract machines. This section highlights the key difference, that of the restricted locations of lambda abstractions, and introduces a new natural semantic specification embodying this restriction.

To focus on the key difference, and in keeping with the previous section, we use a restricted STG language where application and abstraction are only allowed on one argument. We call this language the STG^S language, and its syntax is:

expressions:	e	::=	let $\{x_i = lf_i\}_{i=1}^n$ in e	
		|	$x\ y$	
		|	x	
		|	c *vars*	
		|	case e of *alts*	
lambda-forms:	*lf*	::=	$vars_f \setminus \pi\ x_a.e$	
variables:	*vars*	::=	$\{x_1, \ldots, x_n\}$	$(n \geq 0)$
alternatives:	*alts*	::=	$\{c_j\ \vec{y_j} \to e_j\}_{j=1}^n$	

We have introduced two new syntactic categories, one for convenience (*vars*, representing a possibly empty sequence of variables), and one which is peculiar to the STG machine, a lambda-form (*lf*). A lambda-form replaces the notion of a lambda abstraction, and as can be seen from the syntax, restricts the position of lambda abstractions. We write {} if x_a is empty. In a lambda-form, if x_a is non-empty, then the lambda-form represents a lambda abstraction, x_a being the variable which is abstracted. $vars_f$ represents the free variables of the abstraction. Either x_a or $vars_f$ (or both) may be absent. These free variables play no rôle in the denota-

tional semantics of the language, and are only of operational significance. In addition, a lambda-form has an update flag, π, which plays a rôle when we consider the language operationally. The flag may be set to either u indicating that the bound expression will be updatable, or n indicating that it will not be updatable. We will discover that this is *not* just an optimisation but also necessary for termination, as is discussed in section 4.1.

We may also have a lambda-form which has no argument and only free variables. This corresponds to an ordinary let bound expression (albeit that it cannot be a lambda abstraction) of the basic language. Again, we assume that constructors are saturated.

The most notable differences enforced by the syntax of this language, compared to the basic language, are that:

- The function body of an application must itself must be a simple variable, and not an arbitrary expression.
- Lambda expressions cannot occur in arbitrary places. Instead, lambda expression can only occur let-bound.

Consequences of the restricted syntax

The basic language allows arbitrary expressions in the body of an application:

$$e ::= e\ x \mid \ldots$$

whereas the STG^S language only allows applications of variables:

$$e ::= f\ y \mid \ldots$$

where f and y are variables. A function body other than a variable is not allowed, and so has to be heap allocated by a let binding. This restriction goes hand in hand with the allowed location of lambda abstractions. Since lambda abstractions can only be let bound, and let bindings are heap allocated, this implies that all of the function bodies will be heap allocated. Recall that lambda abstractions are also values – the previous observation implies that if a variable is to be evaluated to a lambda abstraction (signifying a value), then it will instead be evaluated to a variable pointing to the lambda abstraction. That is, we 'detect a value via a variable'. An important intuition is that if an expression is to be evaluated to a lambda abstraction, then it can be evaluated to a variable pointing to the lambda abstraction *since all the lambda abstractions have to be let-bound* and thus will appear in the heap. This hints at the graph reduction ancestry of the STG machine.

We postulate then:

> The STG machine considers a *variable bound to a lambda abstraction* as a value, as opposed to the notion of a *lambda abstraction* being a value. If this concept is incorporated into the natural semantics, an STG machine can be derived.

The rest of this section shows that this is indeed true.

The above notion 'detects' a value sooner, in the sense that we do not wait until a value is rebound in the heap and start executing the value (as in rule Var), but rather stop when we see that we are pointing to it. Note however, that the STG machine does not employ the same concept in its handling of constructors (see (Peyton Jones 1992) for details), and manipulates constructors just as in our basic language semantics.

Thus, we propose a new semantics which instead of evaluating an expression like this:

$$\dfrac{\dfrac{\dfrac{}{\Gamma : \lambda q.q \Downarrow \Gamma : \lambda q.q}\ Lam}{\Gamma[x \mapsto \lambda q.q] : x \Downarrow \Gamma[x \mapsto \lambda q.q] : \lambda z.z}\ Var}{\Gamma : \text{let } x = \lambda q.q \text{ in } x \Downarrow \Gamma[x \mapsto \lambda q.q] : \lambda z.z}\ Let$$

evaluates it like this:

$$\dfrac{\dfrac{}{\Gamma[x \mapsto \lambda q.q] : x \downarrow \Gamma[x \mapsto \lambda q.q] : x}\ Lam_S}{\Gamma : \text{let } x = \lambda q.q \text{ in } x \downarrow \Gamma[x \mapsto \lambda q.q] : x}\ Let_S$$

where we use the notation $\downarrow$ to denote the new relation.

That is, the new derivation does not use a Var rule because of the early detection of the lambda abstraction which we have moved to the new Lam rule. We thus propose that we collapse the rules Var and Lam for the case when a variable is pointing to a lambda abstraction, giving us the behaviour illustrated above. Since we handle the two types of values(constructors and abstractions) differently, we propose the introduction of two Var rules – one for each. Figure 2 suggests a new natural semantics for the extended language based on these observations. The rules follow those in Figure 1 closely, except that in many places we halt when we have a variable bound to a value. The two ways of handling values is reflected in rules Var_S and $VarC_S$.

Aside: We should duplicate rules App_S, Let_S and $Case_S$, so that they too may return constructors as results. As it stands, they can only return pointers to abstractions. This duplication just complicates our presentation, and presents no technical difficulties(see the technical report). It also have no impact on the derived abstract machine.

The semantics presented in Figure 2 needs expressions to be in the form described by the syntax of the extended language. We call a language which has these restrictions λ-normalised. We write the λ-normalisation of an expression e as e^*, and it is defined as follows:

$$\begin{array}{lcl}
x^* & = & x \\
(\lambda x.e)^* & = & \text{let } y = \lambda x.e^* \text{ in } y \\
(\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)^* & = & \text{let } \{x_i = \eta_i\}_{i=1}^n \text{ in } e^* \\
\multicolumn{3}{r}{\textit{where} \begin{cases} \eta_i = \lambda y.e_i'^* & \text{if } e_i \equiv \lambda y.e_i' \\ \eta_i = e_i^* & \text{otherwise} \end{cases}} \\
(e\ x)^* & = & \begin{cases} e\ x & e \text{ a variable} \\ \text{let } y = e^* \text{ in } y\ x & \text{otherwise} \end{cases} \\
(c\ x_1 \ldots x_j)^* & = & c\ x_1 \ldots x_j \\
(\text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n)^* & = & \text{case } e^* \text{ of } \{c_j\ \vec{y_j} \to e_j^*\}_{j=1}^n
\end{array}$$

Note that all introduced variables must be fresh.

Values

Using the natural semantics, we can now state that the meaning of a closed expression e is given by either:

$$\{\} : e \downarrow \Delta[u \mapsto \lambda y.e'] : u$$

or

$$\{\} : e \downarrow \Delta : c\ p_1 \ldots p_j$$

In the first case we can consider the lambda abstraction as the actual result, and in the second the constructor. Indeed, the semantics in Figure 1 will yield one of these (as shown below).

Although the new semantics is less elegant than the old, it does hint of a more operational nature and some conscious design decisions have been made:

- The old semantics cannot 'early-detect' a lambda expressions as a value, as illustrated in the examples above; but this is just a tradeoff: on the one hand the new semantics does not need a 'lambda' instruction, as we will never hit a lambda in the control; on the other, we have introduced an extra check on the value in the heap (is it a lambda abstraction or not).
- The old semantics needs less heap allocation as normalising invariably leads to a greater number of let bindings and thus heap allocations. However, the normalising allows for the early detection of values. A machine capable of application and abstraction of multiple arguments would decrease the heap allocation, and motivates extending our semantics as presented in section 5.
- The new semantics increases the number of times we have to check the type of a closure in the heap. For instance, rule Var_S needs to check that e is not a lambda abstraction.

In the rest of the paper we will assume that the result of evaluating the root expression is a lambda abstraction and not a constructor. All of the propositions below hold in both cases.

Correctness

We can prove that λ-normalisation does not change the meaning of an expression. That is:

Proposition 3 (Correctness of λ-normalisation)

$$\{\} : e \Downarrow \Delta : w \text{ iff } \{\} : e^* \Downarrow \Theta : w$$

Finally, we can state the correctness lemma. The new semantics is correct in the sense that if the original semantics evaluated an expression to a value, then the new one will evaluate the expression to a variable bound to the same value. For λ-normalised terms then, we prove:

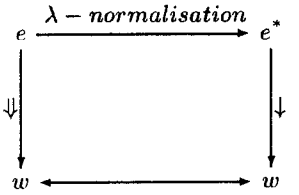
Proposition 4 (Correctness of $\downarrow$)

$\Gamma : m \Downarrow \Delta : w$ *iff* $\Gamma : m \downarrow \Delta'[u \mapsto w] : u$ *where* $\Delta'[u \mapsto w] \stackrel{\alpha}{=} \Delta$

(Note that u may point to an α-renaming of w)

We write $\Delta \stackrel{\alpha}{=} \Lambda$ to indicate that both heaps map the same range to α-equivalent expressions.

Given a closed expression e, proposition 3 tells us that the value of e is unchanged by normalisation. Proposition 4 tells us that for normalised expressions, the value of an expression is the same under both reduction systems (beyond the fact that we result in a pointer to the value instead of the value itself). We have thus shown:



Since propositions 3 and 4 provides a soundness and completeness proof for normalised terms, we have indirectly a proof of correctness with respect to the denotational semantics as well, using proposition 1.

As indicated by proposition 2, Sestoft derives an abstract machine $\mathcal{A}$ which simulates $\Downarrow$. The following section (informally) derives an abstract machine $\mathcal{N}$ which can be proved correct in a similar manner with respect to our natural semantics: That is $e \downarrow w$ iff $e \stackrel{\mathcal{N}}{\rightarrow}^* w$.

All together, we then have that if the meaning of a closed expression e is w under $\Downarrow$, then the abstract machine in the following section will derive this value (actually a pointer to it). That is, our basic STG^S machine is correct.

4 Deriving a STG-like machine

A machine can be formally derived from a natural semantic specification by "flattening" the specification: operationally the natural semantics builds a derivation tree relating an expression to its value from the bottom up, whereas an abstract machine computes the value by building a state sequence. The technique used to make an abstract machine is to represent the context of subtrees by a stack.

Consider the *App* rule:

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : w}{\Gamma : e\ x \Downarrow \Theta : w}$$

This says that to prove that $e\ x$ evaluates to w, we must first find a proof that e evaluates to a lambda abstraction. Then, we must evaluate the expression found by substituting the argument in the abstraction. This final value is the value for the entire application.

Flattening this specification will yield two abstract machine instructions. The first will start the computation of e – and push the argument on the stack to remember to start the computation of the substitution when a lambda abstraction occurs. The second will perform the substitution if a lambda abstraction is found with an argument on the stack.

We now, rather informally, use this technique to derive an abstract machine from the semantic specification in Figure 2. A formal proof of correctness using this technique can be found by following (Sestoft 1997) or (Sansom 1994). The first step to deriving such a machine, however, is the incorporation of a slight change to the semantics. In the current semantics, renaming takes place in the variable rule. A much more natural place for this to take place is in the Let_S rule, allowing the renaming to be mimicked by the allocation of fresh heap locations. This development, and the corresponding proof that the semantics is correct with respect to renaming (that is, that no name clashes occur), can be found in the technical report.

We begin by flattening the natural semantics, as described above. A stack is introduced to represent the flattened tree structure, and we write $\Gamma\ e\ S$ to represent the state of the

	Heap	*Control*	*Environment*	*Stack*	*Rule*
	Γ	$(\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)$	E	S	*let*
$\Longrightarrow$	$\Gamma[p_i \mapsto (e_i, E')]$	e	E'	S	
	$\Gamma[p \mapsto (e', E')]$	x	$E[x \mapsto p]$	S	var_1
$\Longrightarrow$	Γ	e'	E'	$\#p : S$	
	$\Gamma[p \mapsto (\lambda y.e', E')]$	x	$E[x \mapsto p]$	$\#p_u : S$	var_2
$\Longrightarrow$	$\Gamma[p_u \mapsto (\lambda y.e', E')]$	x	$E[x \mapsto p]$	S	
	Γ	$f\ x$	$E[x \mapsto p_x]$	S	app_1
$\Longrightarrow$	Γ	f	$E[x \mapsto p_x]$	$p_x : S$	
	$\Gamma[p_x \mapsto (\lambda y.e, E')]$	x	$E[x \mapsto p_x]$	$p : S$	app_2
$\Longrightarrow$	Γ	e	$E'[y \mapsto p]$	S	
	Γ	case e of *alts*	E	S	$case_1$
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$c_k\ x_1 \ldots x_{a_k}$	E'	$(alts, E) : S$	$case_2$
$\Longrightarrow$	Γ	e_k	$E[y_{ki} \mapsto p_i]$	S	
	Γ	$c_k\ x_1 \ldots x_{a_k}$	E'	$\#p : S$	var_3
$\Longrightarrow$	$\Gamma[p \mapsto (c_k\ \vec{x}, E')]$	$c_k\ \vec{x}$	E'	S	

In the *let* rule, the variables p_i that replace x_i must be distinct and fresh and not occur in Γ or S. The new environment $E' = E[x_i \mapsto p_i]$. In the $case_2$ rule, e_k is the right hand side of the k'th alternative, and $p_i = E'[x_i]$ for $i = 1, \ldots, a_k$. See section 4.1 for a corrected var_1 rule.

Figure 3: The New Abstract Machine ($\mathcal{N}$) derived from the new natural semantics.

abstract machine in which we are evaluating expression e in heap Γ with stack S.

The rules Lam_S and $Cons_S$ give rise to no abstract machine instructions. Intuitively, no machine operations are needed to leave the heap and expression unchanged.

The Let_S rule gives rise to one abstract machine instruction which binds the values in the heap and executes the expression e:

$$\Gamma\ (\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)\ S \Longrightarrow \Gamma[p_i \mapsto e_i]\ (e)\ S$$

The Var_S rule gives rise to two instructions, one to initiate the evaluation of e (and push a marker x onto the stack) and the other to rebind the resulting value (if we find a marker on the stack):

$$\Gamma[x \mapsto e]\ (x)\ S \Longrightarrow \Gamma\ (e)\ (\#x : S)$$

$$\Gamma[x \mapsto \lambda y.e]\ (x)\ (\#z : S) \Longrightarrow \Gamma[z \mapsto \lambda y.e]\ (z)\ S$$

The $VarC_S$ rule also gives rise to two instructions. The first, to initiate the evaluation of e, is identical to that generated by the Var_S rule. The second tells us to perform an update if we have a marker on the stack and *constructor* in the control:

$$\Gamma\ (c\ x_1 \ldots x_j)\ (\#z : S) \Longrightarrow \Gamma[z \mapsto c\ x_1 \ldots x_j]\ (c\ x_1 \ldots x_j)\ S$$

The App_S rule gives rise to two instructions. The first should evaluate the body of the function (and place the argument on the stack), and the second should perform the substitution (when it finds a lambda abstraction and an argument on the stack).

$$\Gamma\ (f\ x)\ S \Longrightarrow \Gamma\ f\ (x : S)$$

$$\Gamma[v \mapsto \lambda y.e]\ (v)\ (x : S) \Longrightarrow \Gamma\ (e[x/y])\ S$$

The $Case_S$ gives rise to two instructions. The first evaluates the expression to a constructor, and the second chooses the correct alternative:

$$\Gamma\ (\text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\})\ S \Longrightarrow \Gamma\ e\ (\{c_j\ \vec{y_j} \to e_j\} : S)$$

$$\Gamma\ (c_k\ x_1 \ldots x_{a_k})\ (\{c_j\ \vec{y_j} \to e_j\} : S) \Longrightarrow \Gamma\ e_k[x_k/y_k]\ S$$

The above machine can be further refined by introducing an environment to model the substitutions taking place. An environment, E, maps variables to heap locations, and can be thought of as a delayed substitution which is only applied when we meet a variable in the control. We thus replace substitutions $e[p/y]$ by a pair $(e, [y \mapsto p])$ representing the fact that within e, y actually points to p. Sestoft (1997) discusses this transformation in more detail. We write $E[x]$ for the value of x under E.

The resulting machine appears in Figure 3.

Even though we are using a restricted STG language, this machine looks much less like the machine described by Sestoft, and very much more like the STG machine. The three rules for the variables (var_1,var_2 and app_2) correspond to the famous STG *Enter* instruction, which enters a closure. Rules *let*, app_1 and $case_1$ correspond to the STG *Eval* instruction and var_2 to the *ReturnCon* rule. The machine described by Sestoft operated on our basic language, which has unrestricted lambda abstractions. As a consequence, his machine has instructions which fire if a lambda abstraction occurs in the instruction stream which make it appear quite different to ours. We have of course, just shown that they essentially do the same thing, though somewhat differently.

4.1 The Neglected Side-Conditions

Recall that the STG language in section 3 annotates lambda-forms with either u or n, indicating whether they are updatable or not. Peyton Jones (1992) states: "It is clearly safe to

set the update flag of every lambda-form to u, thereby updating every closure", and goes on further to explain that an obvious optimisation would be to set the flag to n for 'manifest functions' (which we read here as lambda abstractions), as they are already in head normal form. As support for this hypothesis, the STG machine itself has no rule for an updatable lambda abstraction. This is just as well: if it wasn't for this reasoning and action the STG machine would not work. This optimisation is a prerequisite for correct termination! We argue this point below.

The machine illustrated in figure 3 does not work, as we have forgotten to implement the side conditions of the natural semantics rules Var_S and $VarC_S$. These state that action should only occur if the variable is bound to a non-lambda term. The abstract machine instruction var_1 above does not take this into account, and as such, the machine stops in an erroneous state with a non-empty stack[1]. The corrected abstract machine $\mathcal{N}'$ should only fire rule var_1 if a non-lambda expression is bound. This corresponds to the STG rule described in Peyton Jones (1992) which fires only if the variable is bound to an updatable lambda form and lambda-abstractions are never marked updatable.

We argue that the resulting machine *is* the STG machine, albeit restricted to one argument. Here, we refer to the STG machine as described in (Peyton Jones 1992):

1. An apparent difference is the handling of the environment. All closures need to hold a mapping of variables to heap addresses. In our machine this is represented with the ever present E. In the STG machine, this is represented by the combination of the free variables, vs, and their values, ws_f. Indeed, the STG machine *trims* the environment to contain just those data necessary (the values of the free variables). The trimming is an optimisation, and a similar strategy can be implemented in our machine, as described by Sestoft.

2. The STG machine often looks up the values of the variables in the environment before using them in a later instruction, as opposed to passing the environment through to the later instruction. See for instance STG rules 1 and 2, where p_f is first looked up in the environment. Semantically, rule 1 could have just as well pass the entire environment together with f when executing the *Enter*, and look up f's value in the *Enter* rule. Thus both approaches are equivalent.

3. The STG machine has updatable and non-updatable closures. In our terminology, the updatable closures correspond to expressions which are not lambda-abstractions, and the non-updatable to those which are.

4. The final difference is in the tagging of the code component with *Enter*, *Eval* or *ReturnCon*. This provides no problems, except for the constructor case (we may need to *Eval* a constructor instead of just executing *ReturnCon*). This, together with the premature lookup in the environment explained in (2) above, explains the extra STG rule 5.

We thus have that our rules let, $case_1$, $case_2$ and var_3 correspond with STG rules 3, 4, 6 and 16, while rules app_1, var_1, var_2 and app_2 correspond with the STG rules 1, 15, 17 and 2 given in (Peyton Jones 1992). The full STG machine also has rules for handling integers, which explains the missing numbers. Sestoft describes a very similar way of handling numbers for his abstract machine (inspired by the unboxed representations which the STG machine uses), and so this development presents little difficulty and sheds no light on the STG machine itself.

Following Sestoft it can be shown that:

Proposition 5 (Correctness of the $\mathcal{N}'$ Machine) *Assume w is a value. Then:*

$$(\Gamma, e, [], []) \overset{\mathcal{N}'}{\rightarrow}^* (\Delta[p \mapsto w], p, [], []) \text{ iff } \Gamma : e \Downarrow \Delta[u \mapsto w] : u$$

That is, the single-stack STG machine is correct with respect to the natural semantic specification given in Figure 1.

5 Allowing multiple arguments in abstractions and applications

We now propose an extension of Launchbury's semantics to allow the abstraction and application of multiple arguments. At the moment, if we wish to write the expression $\lambda xy.y$ in our basic language, then we have to write instead the expression let $t = \lambda y.y$ in $\lambda x.t$. As a consequence of this, any abstract machine derived from the semantics will perform a heap allocation (due to the let statement) when executing the above statement. In this section we propose an extension of the semantics given in figure 1 to solve this.

The abstract syntax of the language, which we christen the *extended* language, is given by:

$$\begin{array}{lrll} \text{expressions:} & e & ::= & \lambda \vec{x}^n.e \\ & & | & e\ \vec{x}^n \\ & & | & x \\ & & | & \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \\ & & | & c\ x_1 \ldots x_j \\ & & | & \text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n \end{array}$$

We use the notation, $\vec{x}^n$, to indicate a vector of n arguments.

A few examples

To guide the development of the new semantics, we look at a few examples. Of course, we only expect the semantics to change for the *Lam* and *App* rules, which use the extension. In the following, we assume that w is some arbitrary value, and ignore the contents of the heap.

Having multiple arguments has as consequence that we can now pass too many or too few arguments to a lambda abstraction. In our first example, we look at an under-applied application (an application which is not given enough arguments), given by expression $e \equiv f\ w$ executed in a heap in which f is bound to the expression $\lambda ab.a$. We would expect e to evaluate to a lambda abstraction expecting one argument as the final value, as f will evaluate to a lambda abstraction of arity two, which, after substituting the only argument, will end up with the required left over abstraction. That is, we want:

$$\frac{f \Downarrow \lambda ab.a \quad \lambda b.a[w/a] \equiv \lambda b.w \Downarrow \lambda b.w}{f\ w \Downarrow \lambda b.w}$$

[1] Consider executing let $x = \lambda z.z$ in x.

$$\Gamma : \lambda\vec{x}^n.e \Downarrow \Gamma : \lambda\vec{x}^n.e \qquad Lam_E$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda\vec{y}^n.\lambda\vec{z}^m.e' \quad \Delta : \lambda\vec{z}^m.e'[\vec{x}^n/\vec{y}^n] \Downarrow \Theta : w}{\Gamma : e\ \vec{x}^n \Downarrow \Theta : w}\ m > 0 \qquad App_E$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda\vec{y}^m.e' \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \Downarrow \Theta : w}{\Gamma : e\ \vec{x}^n \Downarrow \Theta : w}\ n \geq m \qquad App'_E$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto w] : \hat{w}} \qquad Var_E$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \Downarrow \Delta : w}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \Downarrow \Delta : w} \qquad Let_E$$

$$\Gamma : c\ x_1 \ldots x_j \Downarrow \Gamma : c\ x_1 \ldots x_j \qquad Cons$$

$$\frac{\Gamma : e \Downarrow \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \Downarrow \Theta : w}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n \Downarrow \Theta : w} \qquad Case_E$$

Note that in rule *App*, e' may actually be a lambda abstraction. We just require that at least n elements can be consumed. Rule *App'* substitutes all available arguments, and applies the resulting expression to the arguments not thus consumed.

Figure 4: Natural Semantic specification of Lazy Evaluation

This suggests the general rule:

$$\frac{\Gamma : e \Downarrow \Delta : \lambda\vec{y}^n.\lambda\vec{z}^m.e' \quad \Delta : \lambda\vec{z}^m.e'[\vec{x}^n/\vec{y}^n] \Downarrow \Theta : w}{\Gamma : e\ \vec{x}^n \Downarrow \Theta : w} \ App_E$$

with $m > 0$.

Now for a case where there are just enough arguments. Executing the expression $f\ w\ w$ with the same heap as before, we would expect w as the resulting value as the *App* rule should substitute both arguments. That is, we want the following behaviour:

$$\frac{f \Downarrow \lambda ab.a \quad a[w/a, w/b] \equiv w \Downarrow w}{f\ w\ w \Downarrow w}$$

What if there are too many arguments? We would expect that the application rule would substitute as many as are needed for the abstraction, and then apply the resulting expression to the rest of the arguments. That is, evaluating $f\ w_1\ w_2\ w_3$ should result in the value of applying w_1 to w_3:

$$\frac{f \Downarrow \lambda ab.a \quad a[w_1/a, w_2/b]\ w_3 \equiv w_1\ w_3 \Downarrow w}{f\ w_1\ w_2\ w_3 \Downarrow w}$$

These suggests a new rule:

$$\frac{\Gamma : e \Downarrow \Delta : \lambda\vec{y}^m.e' \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \Downarrow \Theta : w}{\Gamma : e\ \vec{x}^n \Downarrow \Theta : w} \ App'_E$$

which we want to able to apply when $n \geq m$. In the case where we have just enough arguments, an application will not be formed.

The new, multiple argument, semantics

A natural semantic specification of the extended language is shown in Figure 4. These rules are exactly the same as those in Figure 1 except for the two new application rules discussed above, and the Lam_E rule which says that a lambda abstraction of any number of arguments is still a value.

Following section 3, we can define an *argument-normalisation* which maps the extended language into the basic language:

$$\begin{array}{lcl}
x^{\#} & = & x \\
(\lambda x_1 \ldots x_n.e)^{\#} & = & \text{let } a_n = \lambda x_n.e \\
 & & \quad a_{n-1} = \lambda x_{n-1}.a_n \\
 & & \quad \vdots \\
 & & \quad a_1 = \lambda x_1.a_2 \\
 & & \text{in } a_1 \\
e\ x_1 \ldots x_n & = & \text{let } a_1 = e\ x_1 \\
 & & \quad a_2 = a_1\ x_2 \\
 & & \quad \vdots \\
 & & \quad a_n = a_{n-2}\ x_{n-1} \\
 & & \text{in } a_{n-1}\ x_n \\
(\text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e)^{\#} & = & \text{let } \{x_i = e_i^{\#}\}_{i=1}^n \text{in } Ce^{\#} \\
(c\ x_1 \ldots x_j)^{\#} & = & c\ x_1 \ldots x_j \\
(\text{case } e \text{ of } \{c_j\ \vec{y_j} \rightarrow e_j\}_{j=1}^n)^{\#} & = & \text{case } e^{\#} \text{of} \{c_j\ \vec{y_j} \rightarrow e_j^{\#}\}_{j=1}^n
\end{array}$$

The correctness of the new semantics follows as in the λ-normalisation cases:

Proposition 6 (Correctness of argument-normalisation)

$$\{\} : e \Downarrow \Delta : w \text{ iff } \{\} : e^{\#} \Downarrow \Theta : w$$

$$\Gamma[x \mapsto \lambda\vec{y}^n.e] : x \;\downarrow\; \Gamma[x \mapsto \lambda\vec{y}^n.e] : x \qquad Lam_M$$

$$\frac{\Gamma : f \;\downarrow\; \Theta[v \mapsto \lambda\vec{y}^n.\lambda\vec{z}^m.e] : v \quad \Theta : \lambda\vec{z}^m.e[\vec{x}^n/\vec{y}^n] \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : f\ \vec{x}^n \;\downarrow\; \Delta[u \mapsto w] : u}\ m > 0 \qquad App_M$$

$$\frac{\Gamma : f \;\downarrow\; \Delta[v \mapsto \lambda\vec{y}^n.e'] : v \quad \Delta : e'[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n} \;\downarrow\; \Theta : w}{\Gamma : f\ \vec{x}^n \;\downarrow\; \Theta : w}\ n \geq m \qquad App'_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[u \mapsto w, x \mapsto \hat{w}] : x}\ w \text{ a lambda abstraction, } e \text{ not} \qquad Var_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c\ y_1 \ldots y_j}{\Gamma[x \mapsto e] : x \;\downarrow\; \Delta[x \mapsto c\ y_1 \ldots y_j] : c\ y_1 \ldots y_j}\ e \text{ not a lambda abstraction} \qquad VarC_M$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \;\downarrow\; \Delta[u \mapsto w] : u}{\Gamma : \text{let } \{x_i = e_i\}_{i=1}^n \text{ in } e \;\downarrow\; \Delta[u \mapsto w] : u} \qquad Let_M$$

$$\Gamma : c\ x_1 \ldots x_j \;\downarrow\; \Gamma : c\ x_1 \ldots x_j \qquad Cons_M$$

$$\frac{\Gamma : e \;\downarrow\; \Delta : c_k\ x_1 \ldots x_{a_k} \quad \Delta : e_k[x_1/y_{k1}, \ldots, x_{a_k}/y_{ka_k}] \;\downarrow\; \Theta[u \mapsto w] : u}{\Gamma : \text{case } e \text{ of } \{c_j\ \vec{y_j} \to e_j\}_{j=1}^n \;\downarrow\; \Theta[u \mapsto w] : u} \qquad Case_M$$

Figure 5: Natural Semantic specification of Lazy Evaluation for the λ-normalised, extended, language

Deriving the Abstract Machine

We again, informally, derive the abstract machine from the semantics given in Figure 4. We concentrate only on those instructions which have changed as a result of introducing multiple arguments.

As usual $Lam_E, Cons_E$ give rise to no instructions. Rules $Var_E, Case_E, Let_E$ are unchanged by the extension.

Rule App_E gives rise to two rules. One to start the computation of the body and one to do the substitution. Remember that this rule fires when there are not enough arguments to satisfy the bound abstraction:

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$	
	Γ	e	$(\vec{x}^n : S)$		
2)	Γ	$\lambda\vec{y}^n.e$	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$(e[\vec{x}^n/\vec{y}^n])$	S		

Rule App'_E also gives to two rules. The first is the same as the first above of course, and the second is to form the application of the substituted expression with the remaining arguments:

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$	
	Γ	e	$(\vec{x}^n : S)$		
3)	Γ	$\lambda\vec{y}^m.e$	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$(e[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n})$	S		

We only want rule 3 to fire if there are too many arguments on the stack for the abstraction. We can optimise this last rule somewhat, as we already know what do with an application (rule 1) – the application produced by rule 3 will just dump the rest of the arguments back on the stack and enter the new e (see rule 1)). We could just as well have combined it with rule 1 to produce:

3′)	Γ	$\lambda\vec{y}^m.e$	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$e[\vec{x}^m/\vec{y}^m]$	$(\vec{x}^{(m+1)\ldots n} : S)$		

The following section duplicates this development for above semantics modified for the STG machine. We will see there how rules 2 and 3′ can be further optimised, and how the squeeze operator can be defined.

6 The STG machine, naturally

Repeating the argument of section 3, we can modify the semantics in Figure 4 to detect values via the variable. This results in the semantics given in Figure 5.

Deriving the STG machine

Following the previous section exactly, rule App_M gives rise to the same two instructions, except now rule 2 detects the lambda abstraction in the heap instead of in the control. Remember that this rule fires when there are not enough arguments to satisfy the bound abstraction.

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$	
	Γ	e	$(\vec{x}^n : S)$		
2)	$\Gamma[v \mapsto \lambda\vec{y}^n.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$(e[\vec{x}^n/\vec{y}^n])$	S		

Rule App'_M follows similarly.

1)	Γ	$(e\ \vec{x}^n)$	S	$\Longrightarrow$	
	Γ	e	$(\vec{x}^n : S)$		
3)	$\Gamma[v \mapsto \lambda\vec{y}^n.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$e[\vec{x}^m/\vec{y}^m]\ \vec{x}^{(m+1)\ldots n}$	S		

Again, we can optimise the application by incorporating rule 1:

3′)	$\Gamma[v \mapsto \lambda\vec{y}^m.e]$	(v)	$(\vec{x}^n : S)$	$\Longrightarrow$	
	Γ	$e[\vec{x}^m/\vec{y}^m]$	$(\vec{x}^{(m+1)\ldots n} : S)$		

This begs the question: "How do we count the arguments?". An answer would be to just look at the arguments one by one to determine whether they are update markers, end of stack markers, case markers or the variables that we want. However, *we cannot get a case marker following an insufficient number of arguments.* To see this, we would need to have a: case $(f\ g)$ of *alts* which would dump the *alts* on the stack and execute the scrutiniser which will dump the argument g on the stack and enter f. Then we have *args* followed by *alts*. However, the above program is not well-typed, since the type of $f\ g$ has to be a constructor, and not a partial application.

With this knowledge, we can go back to rule 2 and see that it really looks like this (remember, there are not enough arguments to satisfy the abstraction):

$$2') \quad \begin{array}{ll} \Gamma[v \mapsto \lambda \vec{y}^n.e] & (v) \\ \Gamma & (e[\vec{x}^n/\vec{y}^n]) \end{array} \quad \begin{array}{l} (\vec{x}^n : \#p : S) \\ \#p : S \end{array} \quad \Longrightarrow$$

But since $e[\vec{x}^n/\vec{y}^n]$ is actually a lambda abstraction (there weren't enough arguments), the next rule to fire will be the var_2 (see Figure 3) which will do the heap update for p and remove it and re-execute the lambda abstraction. But this is the same as reentering v with the stack frame removed:

$$2'') \quad \begin{array}{ll} \Gamma[v \mapsto \lambda \vec{y}^n.e] & (v) \\ \Gamma[p \mapsto e[\vec{x}^n/\vec{y}^n]] & (v) \end{array} \quad \begin{array}{l} (\vec{x}^n : \#p : S) \\ (\vec{x}^n : S) \end{array} \quad \Longrightarrow$$

This is the famous *squeeze* operation, which squeezes out the update frame.

We can go even one step further. If we consider generating code for this abstract machine, then we will have a slight problem with rule 2'', because it forms new code – that is, we need to generate code for $e[\vec{x}^n/\vec{y}^n]$ – for each partial substitution. We can get around this by replacing it with the following rule:

$$2''') \quad \begin{array}{ll} \Gamma[v \mapsto \lambda \vec{y}^n.e] & (v) \\ \Gamma[p \mapsto v\ \vec{x}^n] & (v) \end{array} \quad \begin{array}{l} (\vec{x}^n : \#p : S) \\ (\vec{x}^n : S) \end{array} \quad \Longrightarrow$$

which has the same semantics – instead of doing the substitution we make the binding re-apply the body to the arguments. This too forces us to generate code, except this time all we have to do is generate a number of code blocks for an *application*, one for each possible number of arguments. If a uniform representation is used when mapping this machine to hardware, then this is not costly at all (Peyton Jones 1992). In *loc. cit*, the representation of the closure is slightly different: let f be some arbitrary variable, then bind p to the closure $(f\ xs, E[f \mapsto v, xs \mapsto \vec{x}^n])$. The code for this can be shared between all partial applications to the same number of arguments. All that is required is a family of such code-blocks, one for each possible number of arguments.

The final derived machine *is* the STG machine.

7 Conclusions and Future Work

There have been many proposals for evaluating functional languages, such as the G-machine (Johnsson 1984), the spineless tagless G-machine (Peyton Jones 1992), the CMC (Lins 1987) and TIM (Fairbairn & Wray 1987). The work of Lins, Thompson & Peyton Jones (1994) has stressed the importance of relating different abstract machines, allowing us to examine the similarities and differences between the machines. In *loc. cit*, the TIM and CMC machines are related (albeit without sharing), and Peyton Jones & Lester (1992) go some way in showing the relationship between the TIM and the G-machine.

We believe that using natural semantic specifications to compare various machines is a rewarding route to take because:

- the specification provides a common ground from which we can compare the characteristics of different machines. Indeed, all of these machines mentioned above have in common that they attempt to reduce a language in a manner adhering to the principles of lazy evaluation, and this is exactly what the semantics describes. They only each do the same thing slightly differently. For instance, it is quite clear from the development in this paper that the hallmark of the STG machine is its characterisation of an abstraction via a variable. Unfortunately we have also seen that the 'common ground' that we use had to be enlarged somewhat by the inclusion of abstractions and applications to handle multiple arguments, but this should not deter us as any extension should still be consistent with the original semantics.

- the specification can be formally mapped to an abstract machine. As we have demonstrated, this mapping effortlessly handles complex notions such as shared partial abstractions, and provides a framework in which we can examine optimisations to the basic abstract machine (such as the *squeeze* operation). The process of generating an abstract machine can even be automated (Diehl 1996).

- the semantics is *minimal*, in the sense that there is no auxiliary machinery such as stacks and environments. It should be possible to refine this minimal model to build each of the abstract machines mentioned above, thus relating each of them, and making clear the design decisions that were made in creating them and the differences between them. We leave this to our future work.

- the semantics is *useful*, as demonstrated in (Turner, Wadler & Mossin 1995). Sansom & Peyton-Jones (1997) have also used a similar semantics to formally prove a profiling tool correct. This paper has established the relationship between Sestoft's and the STG machine, and so these results can now be carried through (formally) to the STG machine.

Related research includes the works of Launchbury and Sestoft mentioned in this paper. Peyton Jones (1992) mentions that the STG machine regards addresses as values, but takes this notion no further. Lins et al. (1994) relates the TIM and CMC machines, though without sharing. Their approach does not appear to provide a basis for proving other machines correct. (Sestoft 1997) shows a relationship between a one-argument lazified TIM and his derived machine. The work of (Douence & Fradet 1996) builds a very rich framework to compare various implementations and optimisations via successive transformations of a base combinator-like language. Their approach is quite different to ours: we have a high-level semantic specification which can only be transformed into the STG machine and which is related to some base natural semantic specification,

whereas they concentrate on transforming one base language into various abstract machines. Further work is necessary to gauge whether their transformation technique could be used to guide us into different semantic specifications and thus different abstract machines. The work of (Meijer 1988) is similar in that it advocates the successive refinement of a denotational semantics to various abstract machines. (Ariola, Felleisen, Maraist, Odersky & Wadler 1995) propose a reduction semantics of lazy evaluation, and further work will concentrate on determining whether the techniques used for creating abstract machines here carry over to this formalism. Both have their own advantages, as demonstrated in (Turner et al. 1995). Other related works include (Seaman & Purushothaman Iyer 1996) and (Seaman 1993). In their syntax, sharing is not explicitly enforced (leading to a slightly more complicated semantics).

We have not concentrated on the mapping between the STG machine and hardware at all, but this had a large impact on the design of the original STG machine (Peyton Jones 1992). It would be interesting to identify important mapping principles and highlight these in the semantics: for instance, the number of stacks or taglessness. This would allow us to derive an abstract machine even closer to the intended underlying architecture. This would also allow us to quantify the tradeoffs made between the extra heap allocation in the STG as opposed to the lambda instruction in Sestoft's machine. Following the methods of (Hannan 1991) it should be possible to produce machine code in a provably correct manner from the abstract machine.

Acknowledgements

The author thanks Daan Leijen, Pieter Hartel, Simon Peyton Jones and Peter Sestoft for their many useful and insightful comments. The author is supported by the Netherlands Computer Science Research Foundation with financial support from the Netherlands Organisation for Scientific Research (NWO).

References

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. & Wadler, P. (1995), A call-by-need lambda calculus, *in* 'Proceedings of the 22nd ACM Symposium on Principles of Programming Languages'.

Diehl, S. (1996), Semantics-directed generation of compilers and abstract machines, PhD thesis, Universität des Saarlandes.

Douence, R. & Fradet, P. (1996), A taxonomy of functional languages implementations, part II: Call-by-name, call-by-need and graph reduction, Technical Report 3050, Institut National de recherche en Informatique et Automatique (INRIA).

Fairbairn, J. & Wray, S. (1987), TIM: A simple, lazy abstract machine to execute supercombinators, *in* G. Kahn, ed., '3rd Functional Programming Languages and Computer Architecture', Vol. 274 of *LNCS*, Springer-Verlag, pp. 34–45.

Hannan, J. (1991), Making abstract machines less abstract, *in* J. Hughes, ed., '1991 Conference on Functional Programming Languages and Computer Architecture (FPCA)', Vol. 523 of *LNCS*, Springer-Verlag, pp. 618–635.

Johnsson, T. (1984), Efficient compilation of lazy evaluation, *in* 'Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction', Vol. 19(6) of *SIGPLAN notices*, ACM Press, pp. 58–69.

Launchbury, J. (1993), A natural semantics for lazy evaluation, *in* '20th Symposium on Principles of Programming Languages (POPL)', ACM Press, pp. 144–154.

Lins, R. D. (1987), Categorical multi-combinators, *in* G. Kahn, ed., '3rd Functional Programming Langauges and Computer Architecture', Vol. 274 of *LNCS*, Springer-Verlag, pp. 60–79.

Lins, R. D., Thompson, S. J. & Peyton Jones, S. (1994), 'On the equivalence between CMC and TIM', *Journal of Functional Programming* **4**(1), 47–63.

Meijer, E. (1988), A taxonomy of function evaluating machines, *in* T. Johnsson, S. Peyton Jones & K. Karlsson, eds, 'Proceedings of the workshop on Implementation of Functional Languages', Chalmers University of Technology, Technical Report 53.

Peyton Jones, S. L. (1992), 'Implementing lazy functional languages on stock hardware: The STG machine', *Journal of Functional Programming* **2**(2), 127–202.

Peyton Jones, S. L. (1996), Compiling Haskell by program transformation: a report from the trenches, *in* H. R. Nielson, ed., '6th European Symposium on Programming (ESOP'96)', Vol. 1058 of *LNCS*, Springer, pp. 18–44.

Peyton Jones, S. L. & Lester, D. (1992), *Implementing Functional Languages: A Tutorial*, Prentice Hall International Series in Computer Science, Prentice Hall.

Sansom, P. M. (1994), Execution Profiling for Non-strict Functional Languages, PhD thesis, University of Glasgow.

Sansom, P. M. & Peyton-Jones, S. L. (1997), 'Formally-based profiling for higher-order functional languages', *ACM Transactions on Programming Languages and Systems* **19**(2), 334–385.

Seaman, J. (1993), An Operational Semantics of Lazy Evaluation for Analysis, PhD thesis, Pennsylvania State University, Department of Computer Science and Engineering.

Seaman, J. & Purushothaman Iyer, S. (1996), 'An operational semantics of sharing in lazy evaluation', *Science of Computer Programming* **27**, 289–322.

Sestoft, P. (1997), 'Deriving a lazy abstract machine', *Journal of Functional Programming* **7**(3), 231–264.

Turner, D., Wadler, P. & Mossin, C. (1995), Once upon a type, *in* 'International Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 1–11.

The Spineless Tagless G-machine

Simon L Peyton Jones and Jon Salkild
University College London

Abstract

The Spineless Tagless G-machine is an abstract machine based on graph reduction, designed as a target for compilers for non-strict functional languages. As its name implies, it is a development of earlier work, especially the G-machine and TIM.

It has a number of unusual features: the abstract machine code is rather higher-level than is common, allowing better code generation; the representation of the graph eliminates most interpretive overheads; vectored returns from data structures give fast case-analysis; and the machine is readily extended for a parallel implementation.

1 Introduction

Quite a few abstract machine designs for non-strict functional languages have been presented in recent years; examples include the G-machine (Augustsson [1987]; Johnsson [1987]), TIM (Fairbairn & Wray [1987]), the Spineless G-machine (Burn, Peyton Jones & Robson [1988]), the Oregon G-machine chip (Kieburtz [1987]), the Categorical Abstract Machine (Cousineau, Curien & Mauny [1985]), and the CASE machine (Davie & McNally [1989]). At first these appeared as isolated "islands" but it has now become clearer how the various designs relate to each other within a large design space. It is for this reason that we chose a name for our machine which reflects its derivation, rather than inventing a new and unrelated name. This global understanding is very useful, because it allows us to construct a variety of "new" machines by combining features of existing ones. (An attempt to codify this understanding is made by Meijer (Meijer [1988]).) Indeed, the Spineless Tagless G-machine was constructed partly in this way.

This paper follows in the tradition of earlier papers, by exploring a particular local area of the design space, which exhibits a number of unusual features:

- The abstract machine code is rather higher-level than usual, allowing the code generator much more scope to generate good code.
- The graph is represented in a way which almost eliminates interpretive overheads, during both execution and garbage collection.
- Data structures are represented rather compactly, and case-analysis is particularly fast.
- There is scope for exploiting the fruits of both strictness analysis and sharing analysis to improve execution speed.
- A new technique allows pointers and non-pointers to be mixed on the stack without confusing the garbage collector.
- The machine is particularly well-suited for parallel implementations.

Almost all the individual ideas we describe are present in other implementations, but their combination produces a particularly fast implementation of lazy functional languages.

The paper splits into two halves. The first half (Sections 1 to 6) introduces the abstract machine design. The second half (Section 7 onwards) is devoted to a discussion of how we map our abstract machine onto a concrete machine. Some of the techniques are quite entertaining in their own right, but there is a more important reason for describing the mapping in some detail. An abstract machine design is just an intellectual stepping stone between a functional program and a concrete implementation. It must be evaluated both by how well it supports the functional language, but also by how effectively it can be mapped onto a real machine. The fact

that a good mapping must be possible was a driving force in the Spineless Tagless abstract machine design.

The paper is couched in fairly informal terms, but some familiarity with previous abstract machine designs is assumed.

2 Setting the scene

The Spineless Tagless G-machine is capable of executing programs generated from typical high-level non-strict functional languages, such as Miranda[1] (Turner [1985]) or Haskell (Hudak & *et al* [1988]). As usual, however, we assume that the syntactic sugar has been removed, type-checking performed, and lambda-lifting carried out, so that the resulting program is written in a language whose abstract syntax is given in Figure 1.

There is little that is unusual here; the main features are as follows:

- A program consists of a number of supercombinator definitions, each of which defines a function of zero or more arguments, whose body has no free variables except other supercombinators and built-in functions.
- Arbitrary `letrecs` may be embedded within expressions. Prior lambda-lifting ensures that function definitions occur only at the top level, not within `letrecs`.
- Data objects are built by *constructors*, which are distinguished from other functions, and which may only appear applied to their full complement of arguments. (They may, of course, appear partially applied in the source program, but this is easily transformed out.)
- Data objects are taken apart by using case-expressions. These have simple one-level patterns only, and subsume conditionals. The `fail` expression has as its value the value of the default branch of the enclosing case-expression.

Each function definition is compiled first to Spineless Tagless G-machine intermediate code (or Tcode for short), and thence (in our implementation) to M68020 native assembly code for Sun or the GRIP parallel machine. The final assembly and link is performed using the standard Unix tools.

[1] Miranda is a trademark of Research Software Ltd

3 Overview

3.1 An introductory example

We introduce the Spineless Tagless G-machine using a concrete example. Consider the following function definition:

```
compose f g x = f (g x)
```

We will first show the Tcode that is generated from this definition, and then the abstract machine which executes it.

This function would be compiled to the following Tcode:

```
SUPERCOMBINATOR compose [11,12,13] {
    14 := MAKE_CLOSURE [12,13] {
            -- A closure for (g x)
            JUMP 12 [13]
    }
    JUMP 11 [14]
}
```

Arguments to functions are passed on the *argument stack*; each of these arguments is a pointer to a *closure* As usual, a closure consists of a *code pointer* together with zero or more fields giving the environment in which the code should execute; that is, its free variables. A closure is *entered* by loading a pointer to it into a special register called `Node`, and jumping to the code pointer in the closure. The code can then access its free variables via `Node`.

Closures are conventionally used only to represent unevaluated objects, but in our machine *all* objects, whether evaluated or not, are represented by closures. Section 4 deals with the representation of data structures as closures.

On entry to `compose`, therefore, the arguments `f`, `g`, and `x` are on the stack, as shown in Figure 2(a), and the `SUPERCOMBINATOR` instruction names these stack locations `11`, `12` and `13`. Execution then proceeds as follows.

First, the `MAKE_CLOSURE` instruction builds a closure for `(g x)` in the heap, naming it `14`. The first parameter of `MAKE_CLOSURE` is a list of just those intermediate variables which are referred to in the code for the closure, and hence which must be stored in the closure.

Then the `JUMP` instruction performs a tail call to `f`. The tail call involves several steps: a pointer to the closure for `f` is loaded into `Node`; the arguments to `compose` are

```
program      ::= sc_definition*
sc_definition ::= name name* = expression
expression   ::= name expression*
               | constructor expression*
               | letrec (name = expression)+ in expression
               | case expression in alternative+
               | fail
alternative  ::= constructor name* => expression
               | default => expression
```

Figure 1: Source language grammar

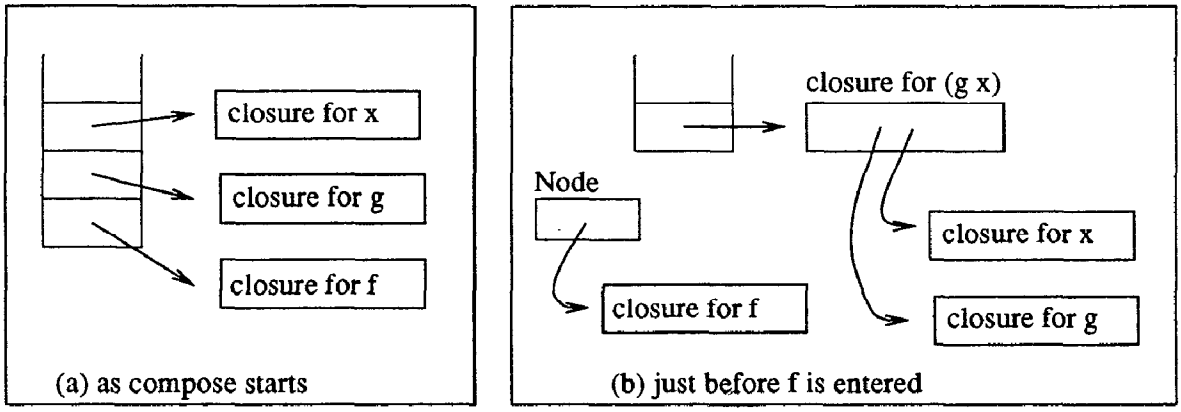


Figure 2: Machine state during execution of compose

removed from the stack; a pointer to the newly-built closure is placed on the stack; and finally the closure for f is entered. The situation just before the jump to the code for f is shown in Figure 2(b).

3.2 The abstract machine code

The overall structure of a Tcode program should be fairly self-explanatory, and a summary of the instruction set is given in Figure 6 at the end of the paper. We make the following general observations.

- Tcode is block-structured rather than linear. For example, the entire code for compose occurs in a block attached to the enclosing SUPERCOMBINATOR instruction. Similarly, the code for the closure is given within the block attached to the MAKE_CLOSURE instruction. The linearisation of the program and the generation of labels required by this process, is postponed to a later stage.

- Tcode does not legislate as to how the rearrangements performed by a function are to take place. Instead, *intermediate variables* are used to name values without specifying exactly where they currently reside, leaving it to the code generator to bind them to concrete locations. There is nothing new in this idea: it is exactly equivalent to the code triples suggested by, for example, Aho, Sethi and Ullman (Aho, Sethi & Ullman [1986, chapter 9]).

  Indeed, Tcode is not far removed from the original program. All that has happened is that the free variables of closures have been made explicit, and a number of explicit intermediate variables have been introduced which makes life more convenient

for the code generator. Furthermore, as will be seen below, the distinction between constructors and functions is made very explicit.

- Tcode is not explicit about how parameters are passed to functions. Instead, the parameters to a function are named by intermediate variables in the `SUPERCOMBINATOR` instruction. Similarly, the `JUMP` instruction implicitly performs all the rearrangements of the machine state required to carry out a tail call.

This approach based on intermediate variables contrasts with many other abstract machines which explicitly use the stack for intermediate values (such as pointers to the closures constructed during the execution of a supercombinator). In order to generate good code for these machines, the code generator has to perform a compile-time simulation of the stack, based on which it can normally elide a large fraction of the stack operations. A really good code generator would figure out a dependency graph for all the operations involved, and generate code to do them as directly as possible. This boils down to recovering from the stack-machine code something very similar to Tcode! We reflect further on the reasons which drove us to such a high-level abstract machine code in Section 10.

3.3 The abstract machine

The abstract machine state consists of the following components:

- The Tcode to be executed.
- A heap, which is a collection of *closures*. A closure consists of a code-pointer, together with zero or more fields required by the code thus pointed to. Typically these will correspond to the free variables of the expression from which the code was compiled.
- An argument stack, containing pointers to closures for the arguments to which the current function is being applied. This corresponds to the "spine" of the expression under evaluation (Burn, Peyton Jones & Robson [1988]).
- An environment, which maps intermediate variables to their values. The values to which these variables are bound may be located in the heap, on the stack, in a machine register, or may just be an immediate constant value.
- A pointer called `Node`, which points to the closure currently under consideration. A closure is *entered* by loading its address into `Node`, and jumping to the code it points to. `Node` therefore plays the role of an environment pointer, since the fields of a closure can be accessed by the code for the closure by indexing from `Node`.
- A special register called `Rtag`, which is used during case-analysis and for arithmetic. It never contains a pointer.

The supercombinator currently being executed may take fewer arguments than are present on the stack. In this case, the supercombinator will eventually return a function which will consume these "extra" arguments. For example, in the function `compose` defined above, suppose there were two further arguments on the stack underneath the closure for `x` in Figure 2. The "extra" arguments simply remain untouched on the stack until required, in exactly the same way as the arguments near the top of the "spine" of an expression represented by binary application nodes (Peyton Jones [1987, chapter 11]).

3.4 Issues raised

The stack-based evaluation model suggested by this example is essentially the same as that of TIM and the Spineless G-machine.

The main difference from TIM (so far) is that closures are self-contained objects consisting of a code-pointer and its free variables, rather than consisting of a code-pointer paired with a pointer to a frame containing (a superset of) its free variables. The main difference from the Spineless G-machine is the emphasis on closures as the abstraction supported by the heap, instead of trees of binary application nodes. We will have more to say later about how closures are represented.

This evaluation model is non-strict but, so far, it only implements *string* reduction. That is, a closure constructed as an argument for a function may be repeatedly evaluated. We address this problem in Section 5.

4 Data structures

The Spineless Tagless G-machine invests heavily in the algebraic data-type model of data structures, which has emerged as the consensus in functional programming languages (for example, Miranda, ML, Haskell).

4.1 An example

Consider the following type declaration and function definition.

```
data Tree = Tip Int | Branch Tree Tree

reflect (Tip n)        = Tip n
reflect (Branch t1 t2) =
     Branch (reflect t2) (reflect t1)
```

The type declaration introduces a new algebraic data type `Tree`, with two *constructors*, `Tip` and `Branch`. The compiler indexes the constructors of the type, starting at zero, thereby assigning to each constructor a small natural number called its *discriminator*. In this example, `Tip` would be given discriminator 0, and `Branch` would have discriminator 1.

The function `reflect` compiles to the following Tcode:

```
SUPERCOMBINATOR reflect [l1] {
   SWITCH Tree { JUMP l1 [] } {
     CASE 0 [l2]:    {
          RETURN_CONSTR Tip [l2]
     }

     CASE 1 [l3,l4]: {
       l5 := MAKE_CLOSURE [l4] {
            JUMPG reflect [l4]
       }
       l6 := MAKE_CLOSURE [l3] {
            JUMPG reflect [l3]
       }
       RETURN_CONSTR Branch [l5,l6]
     }
   }
}
```

The `JUMPG` instruction is just like `JUMP` except that the function to be applied is a named supercombinator.

The `SWITCH` instruction corresponds directly to the case construct, and represents *the only way in which data objects can be taken apart.* `SWITCH` places a return address on the stack, and executes the (arbitrarily complex) code for the expression being scrutinised. In the example, the `JUMP l1 []` instruction will enter the closure given as the first argument to `reflect`. The evaluation of this closure will eventually return a constructor using a `RETURN_CONSTR` instruction.

The `RETURN_CONSTR` instruction returns a data object to the "caller", which must have been a `SWITCH` instruction. It does this by the following steps:

- It builds a closure for the data object, putting a pointer to it in `Node`;
- It puts the discriminator of the constructor in `Rtag`;
- Finally, it returns to the address on top of the stack, which was placed there by the scrutinising `SWITCH`.

Control is thereby returned to the `SWITCH` instruction, which now performs case-analysis on `Rtag`, to decide which branch to execute. Each branch of a `SWITCH` specifies a list of intermediate variables, which are bound to the argument fields of the closure returned in `Node`. Finally, the code given in the selected branch is executed in this environment.

The list of cases in a `SWITCH` need not be exhaustive, a default branch being used to handle all the alternatives which do not have an explicit case. The first argument of the `SWITCH` instruction specifies the data-type being scrutinised, and is used to determine the range of possible values that can be returned in `Rtag`.

There is also a `FAIL` instruction which transfers control to the next enclosing default branch; see (Augustsson [1987]; Peyton Jones [1987, chapter 5]) for the background to this.

4.2 Representing data objects as closures

It should by now be clear that data objects are represented by closures of the same general format as any other closure. What should the code pointed to from such closures do? This question is most easily answered by realising that `reflect` might be applied to just such a closure.

In this case, the `JUMP l1 []` would enter the closure (with no arguments on the stack). It follows that all the code for the data constructor closure needs to do is to load `Rtag` and return to the address on top of the stack. (`Node` is already pointing to the data constructor closure.)

We use this representation uniformly for all data objects. In particular, booleans, lists, characters, integers, and floating-point numbers are all represented in this way, as well as new programmer-defined data types. For example, the type of booleans could be defined like this:

```
data Bool = True | False
```

Conditional expressions are compiled to case-expressions, and thence to SWITCH instructions.

In the type Bool, none of the constructors have any arguments; we say that a data type with this property is *flat* or, equivalently, that it is an *enumeration type*. Characters, integers, and floating-point numbers are all examples of flat types.

One of the beauties of this mechanism is that it turns out to be easy to develop highly-optimised representations for data objects. We discuss this below, when we consider how to map the abstract machine onto a concrete architecture.

4.3 Arithmetic

Representing booleans, lists and characters as algebraic data objects is hardly original, but it is less usual to represent numbers in this way. Integers are objects of type Int, which could be defined thus:

```
data Int = 0 | 1 | 2 | 3 | ...
```

where there are a total of 2^{32} (nullary) constructors. Of course, the compiler knows about integers, so there is no need to write the (rather long) type declaration, but the execution-time representation is exactly what you would get if you did write it out. The effect of entering a closure which represents an integer is simply to load the integer into Rtag and return.

SWITCH can be used to perform case-analysis of integers as you would expect, but more commonly we want to perform arithmetic. Here is an example, which demonstrates the extra instructions we introduce to handle arithmetic:

```
f x y = y*y - x
```

This compiles to the following Tcode:

```
SUPERCOMBINATOR f [l1,l2] {
    u1 := EVAL { JUMP l1 [] }
    u2 := EVAL { JUMP l2 [] }
    u3 := u2 * u2
    u4 := u3 - u1
    RETURN_BASIC u4
}
```

EVAL is similar to SWITCH except that instead of having many continuations, it binds the returned discriminator to the intermediate variable specified in the instruction, continues on to the next instruction [2]. The two instructions following the EVAL perform the arithmetic, and bind u4 to the result. Finally, RETURN_BASIC is similar to RETURN_CONSTR, except that instead of specifying the constructor in the instruction, an intermediate variable containing the discriminator is specified, and the instruction returns a flat data object with the specified discriminator.

Floating-point numbers can be dealt with in a similar way, except that they return their discriminator in a floating-point register Ftag, and each of EVAL, RETURN_BASIC and arithmetic instructions have a floating-point variant. This Ftag return convention means that a SWITCH instruction cannot scrutinise the value of an expression returning a floating-point number, but that is reasonable enough!

Arbitrary-precision numbers are not given any special treatment. They can be programmed using algebraic data types.

4.4 Issues raised

In the scheme we have outlined, RETURN_CONSTR builds a closure in the heap for the data object to be returned, and places a pointer to it in Node. An alternative return mechanism is suggested by the TIM design; RETURN_CONSTR could return the components of the data object *on the stack* (having of course first removed the return address). When control is thus returned to the scrutinising SWITCH, it would perform case-analysis on Rtag as before, but would bind the intermediate variables of the selected branch to the corresponding stack locations rather than to the components of the closure pointed to by Node.

For almost all its life, the Spineless Tagless G-machine has used this TIM-like method. It has a rather nice advantage, namely that *the data object may never be constructed in the heap at all.* Recently, though, we changed the design to the one we have described above, for the following reasons:

- Once evaluated, we believe that data structures are often traversed repeatedly. Under the TIM-like scheme, the code for a closure representing a data object has to load all the fields of the data object onto the stack. The code returned to will

[2] The intermediate variable is prefixed with "u" instead of "l" to remind us that it is bound to a non-pointer.; that is, an "unboxed" value.

then presumably do something with them. Instead, it would be more efficient to move them directly from where they reside in the data object to their final destination. Accessing a field of a data object, by indexing from `Node`, is usually no more expensive than accessing a stack element by indexing from the stack pointer.

- It is quite common that only a subset of the fields of a data object are actually used in the sequel. The most extreme case of this is a selector function, which picks just one field out of the data object. In this case, the work of moving all the other fields onto the stack has been wasted.

- It is quite common for a `case`-expression to select one or two alternatives of a data type, and treat the rest uniformly using a `default` case. With the TIM-like scheme, an exhaustive case-analysis needs to be generated by the compiler regardless, because a single default branch would not know how many items had been placed on the stack by the data object. Under our current scheme, a single default branch suffices.

  Furthermore, in trying to avoid space leaks, it is sometimes useful to be able to evaluate a data object to head normal form, even though no case-analysis is to be performed (at least not at that time). This can be performed by a `SWITCH` which has a `default` branch *only*, so it is just an extreme example of the previous situation.

- Finally, the updating mechanism, which we describe in the next section, turns out to be very much easier and cheaper using the return mechanism we have adopted.

Individually, none of these factors are decisive, but together they convinced us to make the change.

5 Dealing with updates

If a closure is both *shared* and *reducible* (that is, not already in head normal form), then it must be updated with its head normal form if it is ever evaluated — this is the essence of graph reduction. Straightforward implementations of graph reduction normally ensure that this constraint is met by updating the graph after each reduction.

A much more efficient method is used by the Spineless G-machine, and (rather more elegantly) by TIM. The idea is to *update a closure exactly when it has been evaluated to (head) normal form*, and even then only if the closure is shared. It may be hard to determine statically that a closure cannot be shared, but an approximation suffices (provided that it errs only by predicting more sharing than actually occurs!).

How can this moment at which a closure has reached HNF be identified? Suppose that the closure represents a function. When it is entered there will be some arguments on the stack, to which it is being applied. *The closure has been evaluated to HNF exactly when the execution of a supercombinator which requires one or more of these arguments is begun*, because then the closure has been reduced to the form of a partial application of the supercombinator.

This observation suggests the following scheme, which is very similar to that used by TIM and the Spineless G-machine.

- Evaluation normally proceeds without any updates, as described in Section 3.

- The compiler prefixes the code for each closure (that is, the code which is executed when the closure is entered) with a few instructions which set up an *update frame* on the stack.

- The abstract machine is extended by a new register `Sbase`, which points to the topmost word of the uppermost update frame on the stack.

- An update frame consists of a pointer to the closure to be updated, and the old value of `Sbase`.

- The code for each supercombinator begins with a few instructions which determine whether there are enough arguments to satisfy the supercombinator, lying on the stack above the `Sbase` pointer. If not, an update is performed, the the previous value of `Sbase` is restored from the update frame, and the update frame is removed from the stack, by sliding down all the arguments on the stack above the update frame, thus "squeezing it out".

- Finally, the argument-satisfaction check is performed again (possibly triggering another update).

For example, suppose that on entry to a supercombinator of arity three, the machine state is as shown in Figure 3. Since there are only two arguments above `Sbase`, and three are required, an update is triggered.

The update is then performed, as follows. The closure to be updated, which is pointed to from the update

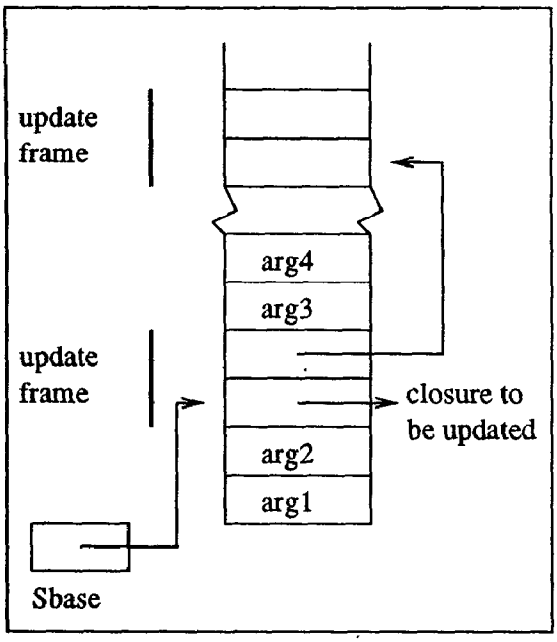


Figure 3: Update frames

frame, is overwritten with an indirection to a new closure (whose form will be described in a moment); the old value of Sbase, which is also in the update frame, is restored; the update frame is squeezed out by moving the arg1 and arg2 pointers down the stack; and the argument-satisfaction check is re-tried. This time it will succeed since four arguments lie on the stack above Sbase.

What is the form of the new closure? It should have the following property: *when entered, it will rebuild the top part of the stack (that above the update frame) in just the way it currently appears, and then jump to the supercombinator whose execution was interrupted by the update.* In our implementation these closures take the form of a standard code-pointer, a field for each argument on the stack above the update frame, and a code-pointer to the supercombinator whose execution was interrupted.

So far we have said that the compiler begins the code for *every* closure with instructions to set up an update frame. If this code prefix is omitted, everything works fine, except that the closure is never updated. If the closure cannot become shared, or is already in normal form, this is a Very Good Thing, because it omits both the construction of the update frame, and the subsequent update operation, without duplicating any work. Thus, if the compiler can perform a static analysis of sharing in the program, it can use this information to optimise the code by selectively omitting the update-setup prefix where it is safe to do so.

5.1 Updating data objects

The above discussion applies to closures whose normal form is a partial function application. However, even when a closure represents an (as yet unevaluated) data object, the same update technique can be used. If we think of the return address as an argument to the constructor, it becomes clear that responsibility lies with the RETURN_CONSTR instruction, which accesses this return address.

Having built the data object in the heap, RETURN_CONSTR should check whether the return address lies above the Sbase pointer, and if not trigger an update operation. This update operation is slightly simpler than before; all it need do is overwrite the closure to be updated with an indirection to the already-constructed data object, restore Sbase, remove the update frame, and repeat the test.

Similarly, the code for constructor closures should check whether the return address lies above Sbase and trigger an update if not.

While this mechanism works perfectly well, it has some annoying features. We believe that data structures are often repeatedly traversed, so a data structure will often be scrutinised by a succession of SWITCH instructions, with no update frames being involved at all. In these cases the test and not-taken jump constitute wasted effort.

These considerations have led us to a simple alternative mechanism. In our implementation, an update frame actually consists of *three* items: the saved Sbase, the pointer to the closure to be updated, and a special "return address" as the uppermost word of the frame, which always points to the same piece of code, called UPD_CONSTR. Now a RETURN_CONSTR instruction can simply return to whatever address is on top of the stack. If an update frame is on top of the stack, then this will return to the UPD_CONSTR code, which performs the update, removes the update frame, and returns to whatever is now on top of the stack; this might either be another update frame or the genuine return address. Figure 4 shows a possible configuration of the stack just before a RETURN_CONSTR instruction performs the return.

The cost of this mechanism is one extra push when an update frame is built, and a pop when it is removed. The benefit is the saving of a comparison and condi-

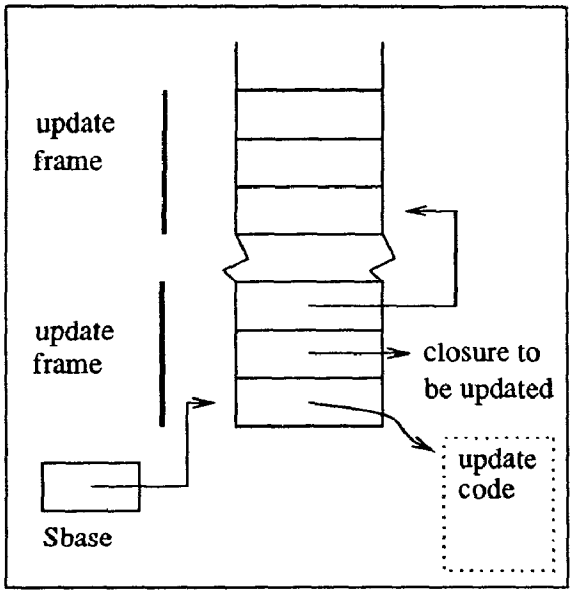


Figure 4: Modified update frames

tional jump in every RETURN_CONSTR instruction and in the code for every constructor closure. We believe that, especially when sharing analysis is used to reduce the use of updates, the balance is in favour of the method we have described, but this remains to be proven.

In this alternative mechanism Sbase plays no role, which allows one further optimisation. If the compiler can tell when it builds a closure that it must represent a data structure, then it can generate code to build a smaller update frame, which does not include the saved Sbase, and a "return address" which does not restore it.

5.2 Issues raised

Suppose that a partial application of a supercombinator causes an update. Once the update has been performed, the update frame has to be removed from the stack, by "sliding down" all the arguments on the stack *above* the update frame, thus "squeezing out" the update frame.

This observation exposes one further advantage of the way in which we return data objects (cf Section 4.4), namely that there is no "sliding" to do, because the object being returned is pointed to by Node. The "sliding" operation only occurs when closures are being updated with partial applications, something which we believe to be very much rarer than updates with data objects. (Certainly, functions are often passed as arguments, but such arguments are frequently recognisable as unsaturated partial applications, and hence do not require updates.)

6 Other Tcode instructions

Suppose that (Cons x y) is passed as an argument to a function. A closure must be built for it, which could be done by a MAKE_CLOSURE instruction, thus:

```
13 := MAKE_CLOSURE [11,12] {
        RETURN_CONSTR Cons [11,12]
}
```

This is rather wasteful, because the closure will be immediately overwritten with a constructor closure when it is entered. Tcode therefore provides an instruction which makes a constructor closure directly, thus:

```
13 := MAKE_CONSTR Cons [11,12]
```

Like the G-machine, the Spineless Tagless G-machine accomodates let- and letrec-expressions quite easily. Consider the following example:

```
f x = let y = g x in
      letrec ys = Cons y zs
      and    zs = Cons x ys
      in ys
```

This is compiled to the following Tcode:

```
SUPERCOMBINATOR f [11] {
    12 := MAKE_CLOSURE [11] {
            -- The closure for y
            JUMPG g [11]
    }
    LETREC [13,14] {
      13 := MAKE_CONSTR Cons [12,14]
            -- The closure for ys
      14 := MAKE_CONSTR Cons [11,13]
            -- The closure for zs
    }
    JUMP 13 []
}
```

A let-expression simply gives rise to a MAKE_CLOSURE instruction, which binds an intermediate variable. A letrec-expression is compiled to a LETREC instruction, which executes the attached Tcode, but in a recursive

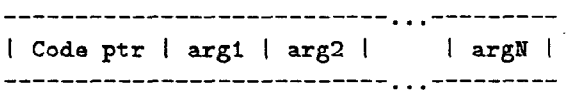
environment. The Tcode attached to a LETREC is always a collection of MAKE_CLOSURE and MAKE_CONSTR instructions, so all LETREC is really doing is constructing a cyclic graph of closures.

7 The heap

The next few sections discuss how the abstract machine is mapped onto a concrete sequential von Neumann computer, beginning with the heap. The heap consists of a collection of closures, so we first describe how an individual closure is represented.

7.1 Representing closures

A *closure* consists of a code-pointer, followed by zero or more argument fields, allocated as a contiguous block of store, thus:



There is no "tag" on the closure, nor garbage-collection mark bits.

A closure is *entered* by jumping to the code-pointer, having loaded the Node register with a pointer to the closure. The code for the closure knows its exact structure, and is able to access the argument fields by indexing from Node.

During the normal running of a functional program, entering is the only operation ever performed on a closure. However, there are various exceptional situations when it is important to be able to perform other operations on a closure. The most important of these situations is garbage collection, when the collector needs to know the size and structure of the closure; that is, which of its argument fields are pointers and which are not. This information is not represented explicitly in the representation we have so far described.

We solve this problem in the following way. An *info-table* is allocated immediately preceding the code pointed to by the code pointer. In our implementation, this info-table contains:

- Enough information to enable the garbage-collector to do its job. In fact we implement this information as two code-pointers, which are described further in Section 7.3.
- Debugging information for inspection by a debugger or trace generator.
- For our parallel implementation, enough information to enable the closure to be flushed into global memory. This, too, is actually implemented as a code-pointer.

In retrospect, this representation is quite similar to that chosen by the Chalmers group for their G-machine implementation. In their system, every heap cell has a one-word tag which points to a table of entry points for the various operations that could be performed on the cell.

Our system differs from theirs in two respects. First, and most important, rather than having a fixed repetoire of "tags", we generate new code for each closure required by the program text, together with its associated info-table. This essentially eliminates the "interpretive unwind" used by the G-machine.

Second, the operation of entering a closure dominates all others, and we perform it using one less indirection than if all entry points were accessed via a table. The effect of this may be small, but it comes for free!

7.2 Allocation

Free space is collected into a single block, and a pointer to one end of it is maintained in a register, Hp. Allocation of a new closure can therefore be performed very efficiently by in-line code, which writes to the address pointed to by Hp, using an auto-decrement addressing mode to simultaneously modify Hp.

A second advantage of this method is that when several closures are allocated in a single basic block, pointers from one to the other can be calculated from the current value of Hp, *including forward pointers to closures not yet constructed* (as is required when generating code for letrec-expressions).

A second register Hlimit points HCLAIM bytes before the end of the free space (HCLAIM is typically around 1000 bytes). The compiler generates code at the start of each basic block which checks whether there is enough heap remaining for all the closures allocated by the block. Since the total required is normally less than HCLAIM, this test normally consists of a single register/register comparison, followed by a jump which is not usually taken. If more than HCLAIM bytes are required, a more expensive check is performed. Notice that (at most) one heap-overflow check is performed per basic block, rather than one per closure allocated.

All these tricks are used in the Chalmers LML compiler (Johnsson [1987]), and it is becoming widely recognised that performing allocation in this general manner is a major contributor to high performance (Appel, Ellis & Li [1988]).

7.3 Garbage collection

Garbage-collection is performed by a two-space copying collector. This involves two basic operations on closures:

- Each live closure must be *evacuated* from from-space to to-space.
- As to-space is scanned linearly, each closure must be *scavenged*; that is, each closure to which it points must be evacuated, unless it has already been evacuated, and the new to-space pointer substituted for the old from-space pointer.

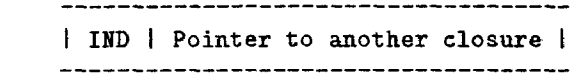
These two operations, evacuation and scavenging, are implemented by two subroutines pointed to from the info-table of each closure. These subroutines know the exact structure of the closure, and therefore can operate without interpretive loops. Furthermore, the scavenge routine knows which of the argument fields are pointers (and hence have to be evacuated), and which are not (and hence must *not* be evacuated).

When a closure is evacuated, it is overwritten by a *forwarding pointer* which points to its new location in to-space, and this new to-space pointer is returned to the caller of the evacuation routine. A forwarding pointer looks much like a bona-fide closure, consisting of a code pointer, `GC_FORWARD`, and a single argument field (the to-space pointer). The important property of the `GC_FORWARD` code pointer is this: when its evacuation routine is called, all that happens is that the to-space pointer is returned to the caller, no data being moved at all. This simple and elegant implementation thereby avoids the requirement to test for a forwarding pointer. (All closures are actually at least two words long, in order to leave enough space for a forwarding pointer.)

Almost all the work of garbage-collection is carried out by the evacuate and scavenge routines of the closures in the heap. A C function does the once-per-collection work of switching spaces and accumulating statistical information.

7.4 Indirections

Indirections are generated by update operations, and they have a particularly efficient representation:



The `IND` code consists of only two instructions: one to load the indirection pointer from the closure into `Node` and a second to enter the new closure.

An indirection can also be "shorted out" rather easily during garbage collection. All that is required is that its evacuation routine jumps to the evacuation routine of the closure to which the indirection points. (The use of "jumps to" rather than "calls" is deliberate — this is a tail call!) Since indirections are thereby never moved into to-space, they don't have a scavenging routine.

8 The stack

The *stack* can be implemented in more than one way. In this section we outline the main issues, and the solution we have adopted.

The simplest way to implement the stack is to map it onto the usual concrete processor stack. There are two major problems with this approach:

- When evaluating arithmetic expressions, it may be necessary to save non-pointers on the stack. Similarly, when evaluating data structures, return addresses are kept on the stack. Such non-pointers on the stack may confuse the garbage collector, which may erroneously treat them as pointers, and try to evacuate the closures they point to. With our "object-oriented" approach to garbage collection, this would cause disaster.
- In a concurrent implementation there may be many tasks, each of which requires a stack. A task may be blocked, which means that its stack must be "frozen" while the processor that was executing it switches to another task. Clearly under these circumstances, *stacks must be heap-allocated objects*, so that stacks can freely be allocated and discarded.

8.1 One stack or two?

The traditional way to deal with the problem of mixing pointers and non-pointers on the stack is to have two (synchronised) stacks, one for pointers and one for non-pointers.

Our implementation originally had two stacks, but we have recently changed to use a single stack, for the following reasons:

- There is only one storage area of uncertain size to allocate.
- There is only one stack to check for overflow. This cannot be done by memory-management hardware if stacks are heap objects.
- A single stack makes things much easier for a concurrent implementation.

Given a single stack, there needs to be some way of distinguishing pointers from non-pointers. This can be done by tagging each stack element with a bit to say whether it is a pointer or not, either by stealing one of the 32 bits in a machine word, or by additionally stacking the tag bit. We believe that this approach imposes an unacceptable overhead on stack operations, and we have developed an alternative technique.

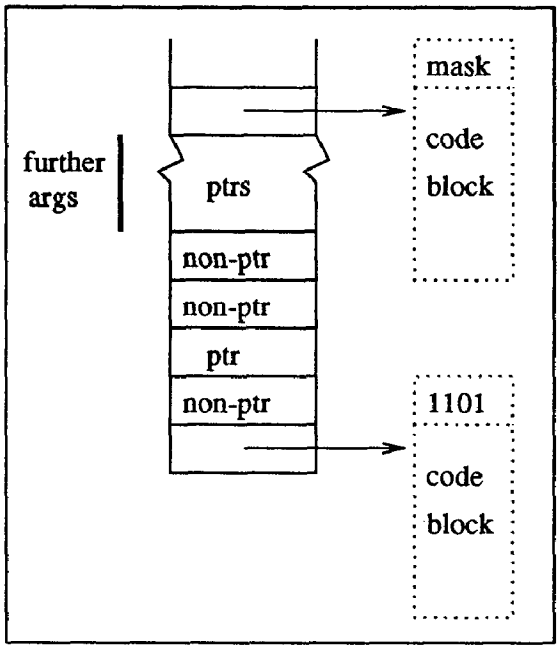


Figure 5: Garbage-collecting the stack

Consider a return address on the stack. It is analogous to the code pointer of a closure, whose fields are the elements of the stack lying below the return address, *in the sense that the return address "knows" which of these elements are pointers and which are not.* Taking the hint from the info-table idea for normal closures, the code generator places a bit-mask in the word immediately preceding the code pointed to by the return address; this bit-mask identifies the non-pointers in the stack elements lying below the return address[3] [4]. Below the stack frame, whose layout is known to the code generator, there may lie some as-yet-unused arguments, but these will always be heap pointers. The garbage collector can evacuate these pointers one by one until it comes across another return address (which it distinguishes by address). Figure 5 shows a possible stack configuration.

This is certainly a relatively complicated solution, but the main overheads are placed where they belong, on the garbage collector rather than on normal evaluation. The main overhead for normal execution is that we cannot use the `call` instruction built in to most processors; instead we have to explicitly push the return address and jump to the function. The other nuisance is that using memory-management hardware to signal stack or heap overflow is completely ruled out, because the exception handler would not have a bit-mask to give the layout of the topmost stack frame.

8.2 The stack as a heap object

Because of our desire to be able to move smoothly from a sequential implementation to a concurrent one, we have chosen to make the stack a heap-allocated object looking like any other closure, even in our sequential implementation. During normal execution, the stack pointer is, of course, held in a register, but when the stack is frozen (for example, before garbage collection, or when a task is blocked), a continuation code pointer is stored on top of the stack, and the stack pointer is stored in a special location in the stack object. When a stack object is entered, it restores the saved stack pointer, and jumps to the continuation point on top of the stack. In addition, of course, a stack object has suitable evacuation and scavenging subroutines in its info-table.

The decision to make stacks into heap objects more-or-less precludes us using memory-management hardware to check for stack overflow. Hence, each function begins

[3] In addition, pointers in the stack which are now dead (ie will never be used again) can be treated in exactly the same way as non-pointers.

[4] If the stack frame is very large, more than one word of bit-mask may be allocated.

by checking that there is enough stack space for the complete execution of the function. We use the same trick with the stack limit register as for the heap limit register, so that the stack overflow check is normally a register/register comparison followed by a non-taken jump. Typically, a single check can cover a larger span of code than the heap-overflow check, because the stack depth is predictable on return from a function call, whereas arbitrary heap allocation may have taken place during the call.

In a sequential implementation it is reasonable to allocate a large stack and to abort the program if the stack overflows. This is not realistic in a concurrent implementation, for two reasons:

- The system would rapidly run out of memory if a large stack object had to be allocated for each individual task, where each such stack had to be big enough to to accomodate the largest possible task.
- When a blocked task is resumed by a processor, its stack may need to be fetched from non-local memory. If the stack is a large object, much of this work may be wasted if the task only runs for a short while before getting blocked again. Breaking the stack into segments allows the system to fetch only the active (top) portion of the stack.

This problem can be addressed by building each stack out of smaller *stack segments*. When the current segment overflows, a new segment can be acquired from the storage manager, part of the current segment copied up into it, and execution resumed.

We have implemented this technique, though space precludes a detailed discussion. The major cost is that of repeatedly checking for stack overflow. There are no checks for stack underflow, because we always break the stack at a return address, and simply arrange that the "return address" in each stack segment does the appropriate things to re-awaken the previous segment.

It is quite interesting to compare this approach with that of Johnsson's $< \nu, G >$-machine (Johnsson [1988]). In the $< \nu, G >$-machine, a closure *is* a stack frame, and contains enough extra space for any temporary variables that may be required. This is rather an elegant approach, and in some ways less complicated than ours. There are two particular reasons why we believe that the segmented-stack approach may prove superior:

- The $< \nu, G >$-machine approach pays some extra run-time overhead when dealing with higher-order functions.
- A stack is one place where spatial locality is fairly certain. Fetching a chunk of stack from non-local store makes efficient use of the communications medium, and the data fetched is quite likely to be useful.

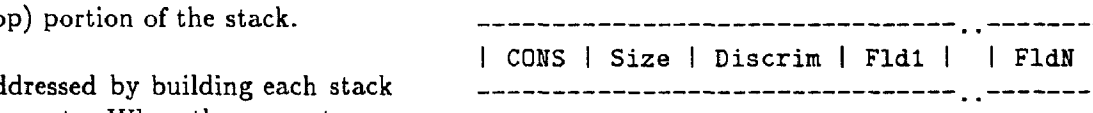
It is much too early to try to make a clear judgement about which method is superior, so it is good that both are being explored.

9 Representing data structures

The "object-oriented" approach to data representation that we have adopted allows an interesting range of optimisations. These break into two main groups: those concerning the format of the closure, and those to do with communication between `RETURN_CONSTR` and `SWITCH`.

9.1 Data constructor closure format

The most general format for a data constructor closure is this:



The `CONS` code just loads the discriminator from the closure into `Rtag` and returns. Its scavenging and evacuation routines use loops based on the `Size` field.

This deals with the general case, but a range of more optimal formats can be created for special cases. For example, a list cons cell is represented like this:

The `CONS_1_2` code loads 1 into `Rtag` and returns; its scavenging and evacuation routines have no loops since they know the exact layout of the closure. Similarly, a special case can be implemented for flat types. Integers are represented like this:

where N is the integer value.

The nice feature is that adding further specialised closure types requires only local changes: some new code in the runtime system, and a modification to the code generator to make it use the new types. [5]

9.2 Vectored returns

So far we have said that RETURN_CONSTR loads Rtag with the discriminator of the constructor and returns to the SWITCH which performs case-analysis on this value. This entails breaking the processor's instruction-fetch pipeline at least twice: once for the return, and at least once for case-analysis. John Hughes suggested to us an idea which reduces this to exactly one pipeline break.

Normally, before the SWITCH executes the code for the expression to be scrutinised it pushes a return address on the stack. Now, suppose instead that it pushes *a pointer to a vector of return addresses.* Now, instead of loading Rtag and returning, RETURN_CONSTR uses the discriminator of the constructor to index into the vector of return addresses pointed to from the stack, and jumps to the appropriate one. Rtag need never be loaded, since the sole purpose of doing so was to communicate with the case-analysis code generated by the SWITCH! (The return-address vector is, of course, allocated statically by the compiler.)

This optimisation is extremely worthwhile. Not only are fewer instructions executed, but also the processor pipeline is broken only once, rather than at least twice for the standard method.

Whether or not a vectored return is used can be decided independently for each datatype. For data types with many constructors, it would prove rather expensive on vector table space; we use non-vectored returns for all datatypes with more than eight constructors.

Of course, a non-vectored return does not preclude the code thus returned to from using a jump table to perform the case analysis. Our code generator decides whether to generate a jump table, or a tree of compare-and-jumps, depending on how densely the jump table would be populated.

10 Code generation

We have implemented a code generator for Tcode, based on the mapping we have described above, which generates Mcode, a simple machine code for a generic von Neumann machine. Mcode is then easily turned into assembly code for a variety of machines, though we have only targetted the M68020 so far. Most of the run-time system is written in Mcode or C [6].

The code generator is written in LML, and is about 3000 lines long (including comments).

10.1 Mapping Tcode onto a concrete machine

As remarked earlier, Tcode is quite a high-level intermediate code, in which most of the details of mapping lexically-scoped intermediate variables onto the concrete machine are left to the code generator. Postponing these low-level decisions until late in the compilation process allows better decisions to be taken. For example:

- Where several closures are allocated within a single basic block, there is no need to keep a separate pointer to each; instead a known offset from the current value of the heap pointer can be used.
- When a JUMP instruction is compiled, the stack must be rearranged ready for the tail call. The simple way to do this is first to build all the new arguments on top of the stack, and then slide them down to overwrite the existing arguments. Our code generator does a full dependency analysis instead, so it can almost always move the new argument directly to its final stack position, ensuring anything thus overwritten is no longer required. This is extremely worthwhile, sometimes halving the number of moves required. Furthermore, when a function calls itself with many arguments unchanged (as happens frequently in lambda-lifted programs), these arguments are not moved at all — a nice bonus!
- When a SWITCH or EVAL instruction is compiled, any volatile intermediate variables must be flushed out onto the stack, because they will not survive across the arbitrary evaluation that may now occur. An example of a volatile variable is one bound to a heap offset, a register, or an offset from the Node register. The code generator decides which variables to flush, on the basis of a live-variable analysis of the code, so a variable which can no longer be referred to is never saved. The nested

[5] This is also true of the Chamlers LML system.

[6] Our Mcode is a modified version of that supported by the Chalmers LML compiler.

structure of Tcode makes it very much easier to carry out this live-variable analysis than if it had been flattened out to a linear sequence of labelled instructions.

10.2 Strictness analysis

The code generator can readily exploit the results of strictness analysis. If a function takes some strict numeric arguments, its code can begin by evaluating these arguments into some standard locations. A fast entry point can then be provided just after this point, which expects the arguments to be passed in evaluated (unboxed) form in these locations. When compiling a call to a known strict global function with all its arguments present, the arguments can be passed in unboxed form to the function's fast entry point.

10.3 Registers considered useless

Lazy languages are rather inimical to register allocators, because whenever a closure is entered (which happens rather frequently) all volatile variables must be flushed out of registers onto the stack. For example, it is seldom worth passing arguments in registers, because the first thing most functions do is to do some pattern-matching, so that all the arguments will be flushed immediately on entry.

The exceptions to this are occasions where the evaluation of numeric closures can be moved to the start of a function body, so that arithmetic on their values can subsequently be performed in registers. Doing this is very important to getting a good `nfib` number, but it is questionable how much impact it has on more typical programs.

One approach to this problem is to test whether a closure is already evaluated before before entering it. If it is unevaluated then the registers can be saved, the evaluation performed, and the registers restored [7]. If it is already evaluated, then it need not be entered at all, and the registers can remain undisturbed. Better still would be a machine architecture in which the top few stack locations *were* the registers. This is not unlike a register-window RISC architecture, except that we would require a continuously-variable window.

[7] In our machine the only way to test whether a closure is evaluated is to perform a range check on its code pointer, having ensured that the code for all normal-form closures was placed in a distinguishable part of memory! If the addressing hardware ignores any of the bits in a code address, one of these bits could be used for the purpose.

11 Performance

Our compiler is a modified version of the Chalmers LML compiler, distributed by Johnsson and Augustsson. This compiler is probably the best currently-available compiler for a non-strict functional language, so it also makes a convenient benchmark for comparison purposes.

So far, only very preliminary performance indications are available, since our compiler has only just sprung into life. Compared with programs generated by the Chalmers LML compiler, our programs seem to run in about two thirds of the time and allocate only 60% as much store. We have not yet implemented sharing or strictness analysis, and our code tests for stack overflow, which the Chalmers code does not.

The more elaborate code generator imposes a some extra compilation cost: compilation times and object code size have increased by 20% compared with the Chalmers compiler.

12 Summary and further work

It can hardly have escaped the reader's notice the the Spineless Tagless G-machine looks really quite like optimised environment-based machines such as Cardelli's FAM (Cardelli [1983]). This is no coincidence, but there are important differences. The use of the stack to hold as-yet-unused arguments is strongly oriented towards the use of higher-order functions, and the way in which updates are performed is closely related to this. The uniform representation of closures for functions and data structures, the use of algebraic data types for *all* data objects, and the vectored-return mechanism are other important differences.

There is much work left to do. We are working hard on developing the sequential version of the Spineless Tagless G-machine compiler to the point where we are happy with its performance. This paper has stated a number of beliefs about the run-time behaviour of functional programs, and we intend to instrument the implementation to give concrete foundation to these beliefs. We also intend to produce a formal definition of the Spineless Tagless G-machine.

We are currently engaged in porting the Spineless Tagless G-machine to the GRIP parallel processor (Peyton Jones et al. [1987]). Space does not permit us to discuss the new issues this raises, but a forthcoming paper will do so.

13 Acknowledgements

We owe a major debt to Thomas Johnsson and Lennart Augustsson. They have allowed us to use their LML compiler as a base for our compiler, which has saved us an enormous amount of work, and many of the ideas described above originated in their work.

Fairbairn and Wray's TIM paper (Fairbairn & Wray [1987]) offered us a radically new perspective on implementation techniques for lazy functional languages, and formed the basis for many of the ideas in our design.

We have discussed the Spineless Tagless G-machine on regular occasions at technical meetings of the GRIP project, and are very grateful to those who have participated in them; especially Chris Clack, Nic Holt, Geoff Burn, John Robson and David Lester.

A Compilation rules for the Spineless Tagless G-machine

This appendix presents simplified compilation rules to Tcode for the grammar given in Section 2, and Figure 6 gives an informal summary of the Tcode instruction set. The following notation is used:

x	is a variable, constructor, or function name
v	is a variable name
c	is a constructor name
f	is a function name
l	is an intermediate variable name

The compilation scheme $\mathcal{F}$ is used to compile each function definition.

$$\begin{aligned}&\mathcal{F}[\![f\ x_1\ \ldots\ x_n = E]\!]\\ &\quad = \texttt{SUPERCOMBINATOR}\ f\ [l_1, \ldots, l_n]\\ &\qquad\quad \{\mathcal{E}[\![E]\!]\ [x_1 = l_1, \ldots, x_n = l_n]\}\\ &\qquad \text{where } l_1, \ldots, l_n \text{ are new intermediate variables}\end{aligned}$$

The $\mathcal{E}$ scheme compiles code to evaluate an expression.

$$\begin{aligned}&\mathcal{E}[\![x\ E_1\ \ldots\ E_n]\!]\,\rho\\ &\quad = l_1 := C[\![E_1]\!]\,\rho;\\ &\qquad \ldots\\ &\qquad l_n := C[\![E_n]\!]\,\rho;\\ &\qquad \mathcal{CALL}[\![x]\!]\,\rho\,[l_1, \ldots, l_n]\\ &\qquad \text{where } l_1, \ldots, l_n \text{ are new intermediate variables}\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{letrec}\ x_1 = E_1\ \ldots\ x_n = E_n\ \texttt{in}\ E]\!]\,\rho\\ &\quad = \texttt{LETREC}\ [l_1, \ldots\ l_n]\ \{\\ &\qquad\quad l_1 := C[\![E_1]\!]\ \rho';\\ &\qquad\quad \ldots\\ &\qquad\quad l_n := C[\![E_n]\!]\ \rho'\\ &\qquad \};\\ &\qquad \mathcal{E}[\![E]\!]\,\rho'\\ &\qquad \text{where } \rho' = \rho[x_1 = l_1, \ldots, x_n = l_n]\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{case}\ E\ \texttt{in}\ alt_1\ \ldots alt_n]\!]\,\rho\\ &\quad = \texttt{SWITCH}\ \{\mathcal{E}[\![E]\!]\,\rho\}\{\\ &\qquad\quad \mathcal{S}[\![alt_1]\!]\,\rho;\\ &\qquad\quad \ldots\\ &\qquad\quad \mathcal{S}[\![alt_n]\!]\,\rho\\ &\qquad \}\end{aligned}$$

$$\mathcal{E}[\![\texttt{fail}]\!]\,\rho = \texttt{FAIL}$$

The C scheme compiles code to construct a closure for an expresssion. $\mathcal{FV}[\![E]\!]$ is the set of free variables of E.

$$\begin{aligned}C[\![v]\!]\,\rho &= \rho(v)\\ C[\![E]\!]\,\rho &= \texttt{MAKE\_CLOSURE}\ \mathcal{FV}[\![E]\!]\ \{\mathcal{E}[\![E]\!]\,\rho\}\end{aligned}$$

$\mathcal{CALL}$ simply generates the correct sort of instruction for a tail call.

$$\begin{aligned}\mathcal{CALL}[\![v]\!]\,\rho\ args &= \texttt{JUMP}\ \rho(v)\ args\\ \mathcal{CALL}[\![f]\!]\,\rho\ args &= \texttt{JUMPG}\ f\ args\\ \mathcal{CALL}[\![c]\!]\,\rho\ args &= \texttt{RETURN\_CONSTR}\ c\ args\end{aligned}$$

The $\mathcal{S}$ scheme compilles code for a case alternative.

$$\begin{aligned}&\mathcal{S}[\![c\ x_1\ \ldots\ x_n \Rightarrow E]\!]\,\rho\\ &\quad = \texttt{CASE}\ discriminator(c)\ [l_1, \ldots, l_n]:\\ &\qquad\quad \{\ \mathcal{E}[\![E]\!]\,\rho[x_1 = l_1, \ldots, x_n = l_n]\ \}\\ &\qquad \text{where } l_1, \ldots, l_n \text{ are new intermediate variables}\end{aligned}$$

$$\begin{aligned}&\mathcal{S}[\![\texttt{default} \Rightarrow E]\!]\,\rho\\ &\quad = \texttt{DEFAULT}:\ \{\ \mathcal{E}[\![E]\!]\,\rho\ \}\end{aligned}$$

B References

AV Aho, R Sethi & JD Ullman [1986], *Compilers - principles, techniques and tools*, Addison Wesley.

SUPERCOMBINATOR *id args body*	Define a new supercombinator *id* with arguments *args* and body *body*.
l := *g*	Bind the intermediate variable *l* to the global constant *g*.
l := MAKE_CLOSURE *freevars code*	Build a closure with free variables *freevars* and code *code*, and bind *l* to it.
l := MAKE_CONSTR *constr args*	Build a data object with constructor *constr* and arguments *args*, and bind *l* to it.
l := MAKE_STRING *string*	Bind *l* to a closure representing the constant string *string*.
LETREC *vars code*	Perform a multiple simultaneous binding of the intermediate variables *vars*. The bindings themselves are given by MAKE_CLOSURE and MAKE_CONSTRUCTOR instructions in *code*.
JUMP *l args*	Tail-call the function bound to *l*, passing the arguments bound to the intermediate variables *args*.
JUMPG *g args*	Tail-call the global function *g*, passing the arguments bound to the intermediate variables *args*.
RETURN_CONSTR *constr args*	Return the data object consisting of the constructor *constr* with arguments *args* to the return address on the stack.
SWITCH *type discrim cases default*	Perform case-analysis on the object of type *type* returned by executing the code *discrim*. *cases* is a list of (discriminator, code) pairs, and *default* gives the code to be executed the data object returned doesn't match any of the discriminators given.
FAIL	Jump to the default code for the innermost enclosing SWITCH, setting the stack to the depth expected by the destination.
u := EVAL *code*	Execute *code*, which will return a value of an enumerated type, and bind *u* to the discriminator thus returned.
RETURN_BASIC *type u*	Return a data object of enumeration type *type*, whose tag is bound to the intermediate variable *u*, to the return address on the stack.
u := BASICOP *op args*	Bind *u* to the result of performing some built-in arithmetic operation to the values bound to *args*.

Figure 6: Tcode instruction set

AW Appel, JR Ellis & Kai Li [June 1988], "Real-time concurrent collection on stock multiprocessors," in *Proc SIGPLAN Conference on Programming Language Design and Implementation, Atlanta*, ACM, 11–20.

Lennart Augustsson [1987], "Compiling lazy functional languages, part II," PhD thesis, Dept Comp Sci, Chalmers University, Sweden.

Geoff Burn, Simon L Peyton Jones & John Robson [July 1988], "The Spineless G-machine," in *Proc ACM Conference on Lisp and Functional Programming, Snowbird*, 244–258.

Luca Cardelli [Jan 1983], "The functional abstract machine," *Polymorphism* 1.

G Cousineau, PL Curien & M Mauny [Sept 1985], "The Categorical Abstract Machine," in *Functional Programming Languages and Computer Architecture, Nancy*, JP Jouannaud, ed., LNCS 201, Springer Verlag, 50–64.

AJT Davie & DJ McNally [Jan 1989], "CASE - a lazy version of an SECD machine with a flat environment," Dept of Computer Science, University of St Andrews.

Jon Fairbairn & Stuart Wray [Sept 1987], "TIM - a simple lazy abstract machine to execute supercombinators," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer Verlag LNCS 274, 34–45.

Paul Hudak & Phil Wadler *et al* [Dec 1988], "Report on the functional programming language Haskell," Dept of Computer Science, Yale University.

T Johnsson [Nov 1988], "The ν-G-machine," Dept of Computer Science, Chalmers University.

Thomas Johnsson [1987], "Compiling lazy functional languages," PhD thesis, PMG, Chalmers University, Goteborg, Sweden.

RB Kieburtz [Oct 1987], "A RISC architecture for symbolic computation," in *Proc ASPLOS II*.

Erik Meijer [Sept 1988], "Generalised expression evaluation," in *Proc workshop on implementation of lazy functional languages, Aspenas*.

Simon L Peyton Jones [1987], *The implementation of functional programming languages*, Prentice Hall.

Simon L Peyton Jones, Chris Clack, Jon Salkild & Mark Hardie. [Sept 1987], "GRIP - a high-performance architecture for parallel graph reduction," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer Verlag LNCS 274, 98–112.

David Turner [Sept 1985], "Miranda - a non-strict functional language with polymorphic types," in *ACM Conf on Func Prog and Comp Arch, Nancy*.