1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center

Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach†, Rob Gardner, Tom Christian, Lucy Cherkasova
Hewlett-Packard Laboratories, †Technical University of Munich (TUM)
{firstname.lastname, brian.j.watson}@hp.com, †daniel.gmach@in.tum.de

Abstract

Recent advances in hardware and software virtualization offer unprecedented management capabilities for the mapping of virtual resources to physical resources. It is highly desirable to further create a "service hosting abstraction" that allows application owners to focus on service level objectives (SLOs) for their applications. This calls for a resource management solution that achieves the SLOs for many applications in response to changing data center conditions and hides the complexity from both application owners and data center operators. In this paper, we describe an automated capacity and workload management system that integrates multiple resource controllers at three different scopes and time scales. Simulation and experimental results confirm that such an integrated solution ensures efficient and effective use of data center resources while reducing service level violations for high priority applications.

1. Introduction

Data centers are inexorably growing more complex and difficult for humans to manage efficiently. Although virtualization provides benefits by driving higher levels of resource utilization, it also contributes to this growth in complexity. Data centers may include both hardware- and software-level virtualization, such as HP's Virtual Connect [1] network virtualization technology, as well as the hypervisor-based VMware ESX Server [2], Citrix XenServer [3], and Virtual Iron [4] products. Each technology offers different control "knobs" for managing the mapping of virtual and physical resources. Continually adjusting these knobs in response to changing data center conditions can minimize hardware and energy costs while meeting the service level objectives (SLOs) specified by application owners. This activity should be automated to help avert the coming complexity crisis and more fully realize the benefits of virtualization.

The purpose of our work is to enable both application owners and data center operators to focus on service policy settings, such as response time and throughput targets, and not worry about the details of where an application is hosted or how it shares resources with others. These details are handled by our resource management solution, so that system administrators can "set it and forget it".

This paper describes three key contributions. First, we propose the 1000 Islands solution architecture that supports automated resource management in a data center. It exploits multiple control knobs at three different scopes and time scales: short-term allocation of system-level resources among individual workloads on a shared server, medium-term live migration of virtual machines (VMs) between servers, and long-term organization of server clusters and groups of workloads with compatible long-term demand patterns. This architecture integrates multiple resource controllers that are implemented using different analytic techniques including control theory, bin packing, trace-based analysis and other optimization methods. The innovation is in leveraging each of these independently and then combining their power. Second, we define specific interfaces for coordinating the individual controllers at run time to eliminate potential conflicts. This includes interfaces for sharing policy information, so that policies do not have to be duplicated among controllers. Finally, we validate the effectiveness of the integrated solution through a simulation study, as well as experimental evaluation on a testbed built from real systems.

Section 2 presents the 1000 Islands solution architecture, and explains how its three controllers are integrated. Section 3 describes the simulation environment and the experimental testbed used to validate the architecture. The performance evaluation results from two case studies are shown in Section 4. Section 5 discusses related work. In Section 6, we conclude and discuss future research directions.

2. Our Solution

The 1000 Islands architecture (shown in Figure 1) consists of three individual controllers operating at different scopes and time scales:

- On the longest time scale (hours to days), **pod set controllers** study the resource consumption history of many *workloads* (WLs), which are applications or their components, and which may run in VMs. This controller determines whether the data center has enough resource capacity to satisfy workload demands, and places compatible workloads onto *nodes* (servers) and groups nodes into *pods* (workload migration domains). A *pod set* can consist of multiple non-overlapping pods.
- On a shorter time scale (minutes), **pod controllers** react to changing pod conditions by adjusting the placement of workloads on nodes.
- On the shortest time scale (seconds), **node controllers** dynamically adjust workload resource allocations to satisfy SLOs for the applications.

The next three subsections describe the three individual controllers, and the last subsection presents how the three controllers are integrated.

2.1. Node controller

A node controller is associated with each node in a pod. It manages the dynamic allocation of the node's resources to each individual workload running in a VM. The node controller consists of two layers: a set of utilization controllers (UCs) for the individual VMs, and an arbiter controller (AC) for the node. Figure 2 shows this layered structure. In all the discussions that follow, we use the term *resource* to refer to a specific type of system resource (e.g., CPU) on a node, although the algorithm can be generalized to handle multiple resources (such as disk I/O or network bandwidth) at the same time.

A utilization controller collects the average resource consumption of each VM from a sensor (see "S" in Figure 2), and determines the required resource allocation to the VM such that a specified utilization target can be achieved. This is done periodically with a control interval of seconds. We define a VM's *utilization* as the ratio between its resource consumption and resource allocation. For example, if a VM's measured average CPU consumption in a control interval is 0.3 of a CPU, and the specified utilization target is 75%, then the utilization controller will drive the CPU allocation for the VM towards 0.4 of a CPU in subsequent control intervals. The utilization target for a workload is driven by its need to meet its application-level SLO. For example, in order for an interactive Web application to meet an average response time goal of 1 second, the average CPU utilization of the VM running the application may have to be maintained at 60%. A feedback controller was presented in [5] to automatically translate application-level SLOs to VM-level utilization targets. This enhanced controller will be integrated into the 1000 Islands architecture later.

All utilization controllers feed the desired resource allocation for each VM (referred to as a *request*) into the arbiter controller, which determines the actual resource allocations (referred to as an *allocation*) to the VMs. If the sum of all requests is less than the node's capacity, then all requests are granted. In addition, the excess capacity is distributed among the VMs in proportion to their requests. On the contrary, if the sum

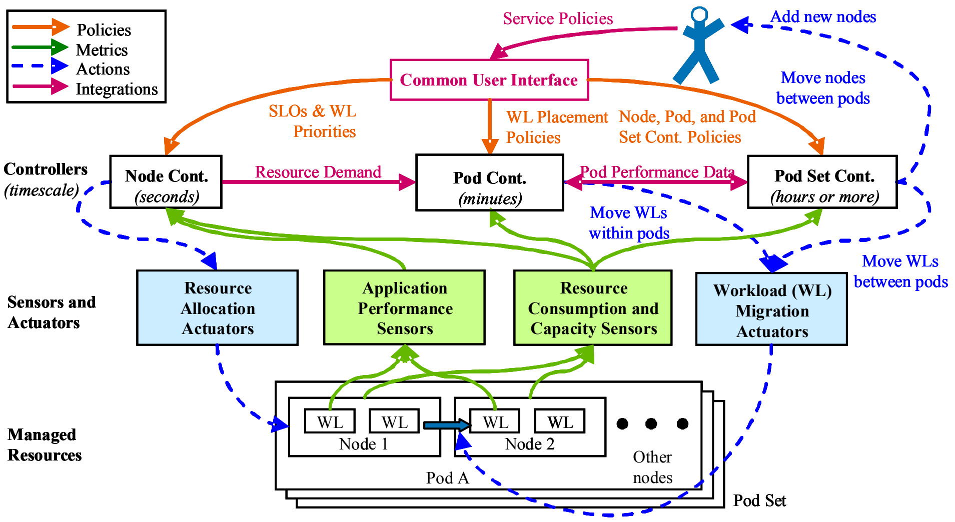


Figure 1. The 1000 Islands solution architecture consisting of node, pod, and pod set controllers

of all requests exceeds the node's capacity, the arbiter controller performs *service level differentiation* based on workload priorities defined in policies by data center operators. In our current implementation, each workload is assigned a priority level and a weight within that level. A workload with a higher priority level always has its request satisfied before a workload with a lower priority level. Workloads at the same priority level receive a percentage of their requests in proportion to their weights. Finally, the allocations determined by the arbiter controller are fed into the *resource allocation actuators* (see Figure 1 or "A" in Figure 2) for the next control interval.

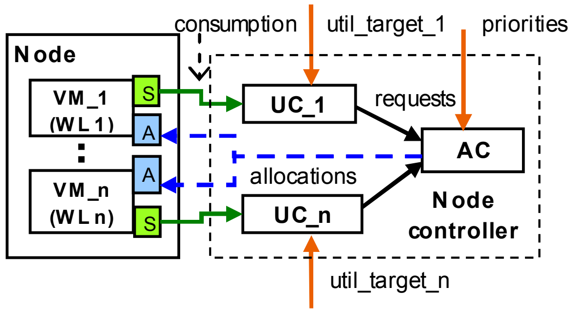


Figure 2. Node controller architecture

2.2. Pod controller

The primary purpose of the pod controller is to react to aggregate CPU requests exceeding a node's capacity, exploiting the fact that they will rarely exceed total pod capacity. In the event that total pod capacity is exceeded, high priority VMs should be favored over low priority ones. The pod controller uses live migration of VMs to move workloads between nodes [6]. In our experiments, moving a 512MB VM takes slightly over one minute from migration initiation to completion, with only sub-second actual downtime. This makes live migration effectively transparent to the workload inside the migrated VM, though the nodes experience transient CPU and LAN overheads. Next, we describe two implementations of the pod controller that are used in our experiments.

CF pod controller

The CF pod controller implementation [7] consists of a simulated annealing algorithm that periodically searches for VM to node mappings in accordance with a node overload avoidance and mitigation policy. Candidate mappings are generated by modeling the effects of a succession of random VM migrations, and are evaluated using a cost function that penalizes mappings that lead to overload conditions on any node. A node is defined as overloaded when the total CPU request or memory consumption of its VMs, plus the hypervisor and the DOM-0 overheads, exceeds the available capacity of the node. Mappings with some headroom are favored to avoid overload or SLO violations. To mitigate overloads that do occur, mappings with fewer high priority VMs per node enable more effective service level differentiation by the node controller. This is done with a non-linear penalty on the count of high priority VMs per node. The best mapping is turned into a migration list and fed into *workload migration actuators* (see Figure 1).

TUM pod controller

The TUM pod controller [8] uses a fuzzy logic feedback control loop. It continuously monitors the nodes' resource consumptions for values that are too low or too high. In our experiments, a node is overloaded whenever its CPU or memory consumption exceeds 85% or 95%, respectively. Furthermore, a pod is lightly utilized if the average CPU or memory consumption of all nodes drops below 50% or 80%, respectively. Our justification for these thresholds is beyond the scope of this paper, but appears in related work [9]. After detecting a lightly utilized or overloaded situation, the pod controller identifies actions to remedy the situation, considering the load situation of all affected nodes and workloads. If a node is overloaded, it first determines a workload on the node that should be migrated, and then searches for another node to receive the workload. These rules also initiate the shutdown and startup of nodes to help reduce power usage within a pod.

2.3. Pod set controller

A pod set controller determines whether a data center pod set has enough resource capacity to satisfy all workloads, and periodically determines compatible sets of workloads to place onto nodes within each pod. Our pod set controller [10] supports capacity planning for pod sets, as well as objectives that consolidate workloads to a small number of nodes or balance workloads across nodes. To accomplish this, it studies the historical resource demands of each workload and assumes that future demand patterns will be similar to past demand patterns. The pod set controller simulates the future resource demand behavior of alternative workload placements and uses an optimization heuristic to determine a placement expected to take best advantage of statistical multiplexing among time-varying workload demands.

2.4. Controller integration

One of our contributions is identifying necessary interfaces between these three controllers, so that they can work well together to achieve fully automated capacity and workload management at a data center scale. The red arrows in Figure 1 indicate these integration points. First, the node controllers must provide estimated resource demands to the pod controller. Otherwise, the pod controller might estimate resource demands that do not agree with the node controllers. If the pod controller's estimates are too low, then it will pack too many workloads onto a node, possibly causing application-level SLO violations. On the other hand, estimates that are too high could trigger an excessive number of overload conditions, and would reduce the power savings that could be achieved by consolidating workloads onto as few nodes as possible.

Second, the pod controller must provide pod performance data to the pod set controller so that the latter can improve the compatibility of workloads it places in each pod, and react to pod overload or underload situations by adding or removing nodes. Third, the pod set controller should provide hints to the pod controller about what will happen in the near future. If a workload's resource demand is expected to increase significantly at a particular time, then the pod controller can prepare in advance by placing that workload on a lightly loaded node. Finally, all three controllers must be configurable through a single user interface, and they must consider the other controllers' configuration parameters. For example, the pod controller needs to know the workload priorities used by the node controllers, so that it does not group too many high priority workloads onto the same node, thus preventing effective service level differentiation. When properly integrated, these controllers automate resource allocation and hide the complexity of resource management from data center operators.

3. Validation of the Solution

In order to validate the design of the proposed architecture and to demonstrate the merits of the integration approach, we have built both a host load emulator and an experimental testbed to perform workload consolidation case studies using real-world resource consumption traces from enterprise data centers. For the work described here, we have used the experimental testbed for evaluating the integration of the CF pod and node controllers in a small-scale pod, and the emulator for evaluating the integration of the pod set and TUM pod controllers in a large-scale pod with a larger number of workloads. This section describes the setup of these two environments.

3.1. Host Load Emulator

Predicting the long term impact of integrated management solutions for realistic workloads is a challenging task. We employ a simulation environment to evaluate a number of management policies in a time effective manner.

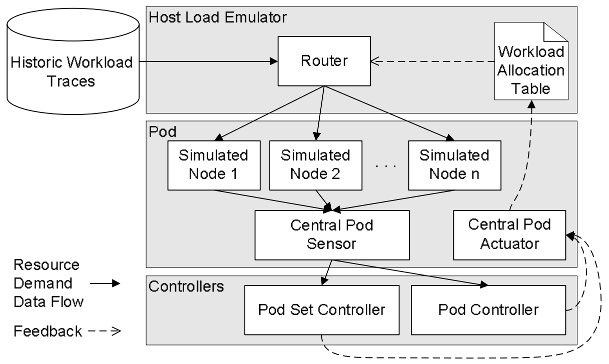


Figure 3. Simulation environment setup

The architecture for the host load emulator is illustrated in Figure 3. The emulation environment takes as input historical workload resource consumption traces, node resource capacity descriptions, pod descriptions, and the management policy. The node resource capacity descriptions include numbers of processors, processor speeds, and physical memory size. A routing table directs the historical time-varying resource consumption data for each workload to the appropriate simulated node, which then determines how much of its aggregate workload demand can be satisfied and shares this time varying information through the *central pod sensor*. The management policy determines how controllers are invoked. Controllers periodically poll the sensor and decide whether to migrate workloads from one node to another, which is initiated by a call to the *central pod actuator*. In our emulation environment this changes the routing table and adds an estimated migration overhead to both the source and destination nodes for the duration of the migration.

Our emulator gathers various statistics, including the frequency and length of CPU and memory saturation periods, node capacity used in terms of CPU hours, and the number of workload migrations. Different controller policies have different behaviors that we observe through these metrics.

3.2. Experimental testbed

Our experimental testbed consists of eight VM hosts, as well as several load generator and controller machines, all interconnected with Gigabit Ethernet. Each VM host is an HP Proliant server consisting of dual 3.2GHz Pentium D processors with 2MB of L2 cache, 4GB of main memory, and SLES 10.1 with a Xen-enabled 2.6.16 kernel. Storage for the VMs is provided by an HP StorageWorks 8000 Enterprise Virtual Array, and the nodes connect to the array via Qlogic QLA2342 Fiber Channel HBA. Each VM is configured with 2 virtual processors and 512MB of memory, and runs SLES 10.1 for best interoperability with the Xen hosts.

We use an Apache Web server (version 2.2.3) as the test application inside each Xen VM. It serves CGI requests, each doing some random calculation and returning the result in HTML. Another eight physical machines are used to generate workload demands on the VMs. These "driver" machines are mostly dual AMD Opteron servers with 1MB of L2 cache and 8GB of main memory, each running Redhat AS4. Each driver machine hosts two instances of a modified version of httperf [11], which can continuously generate a variable number of concurrent HTTP sessions. Each session consists of a series of CPU-intensive CGI requests. In order to reproduce the CPU consumption from the real-world resource consumption traces, we first ran experiments to calibrate the average CPU time used by a CGI request, and then we calculated the CGI request rate to produce a given level of CPU consumption.

The Xen hypervisor interface exposes counters that accumulate the CPU time (or cycles) consumed by individual VMs. The counters are sampled at fixed intervals, effectively yielding a sensor for CPU consumption (i.e., *resource consumption sensor* in Figure 1). Information on the completed transactions, like URLs and response times, is collected on the client side. Xen also exposes interfaces in Dom-0 that allow run time adjustment of scheduler parameters such as the CPU share for each VM (i.e., *resource allocation actuator* in Figure 1). In our experiments, we use the Credit Scheduler as the actuator for CPU allocation, operated in the capped mode, which means that a VM cannot use more than its share of the total CPU time, even if there are idle CPU cycles. This non-work-conserving mode of the scheduler provides a straightforward guarantee on the CPU time allocated to a VM and provides performance isolation among workloads hosted by different VMs. Live VM migration in Xen uses a bounded iterative pre-copy of VM memory from node to node, followed by a stop and copy of remaining or recently dirtied pages [6]. This increases the time between migration initiation and completion, in favor of minimizing VM down time when network connections might be lost.

4. Results from Case Studies

The following subsections discuss performance evaluation results from two case studies. In the first, we used the host load emulation environment to evaluate the pod set and TUM pod controllers. The second case study was done on our experimental testbed using the node and CF pod controllers.

4.1. Integrated pod set and pod controller

In this study, we focus on the use of the pod set controller within a single pod. The evaluation used real-world load traces for 138 SAP enterprise applications. The load traces captured average CPU and memory consumption as recorded every 5 minutes for a three month period. The host load emulator walked forward through this data in successive 5 minute intervals. The nodes in the emulated pod had 8 2.93-GHz processor cores, 128 GB of memory, and two dual 10 Gigabit Ethernet network interface cards for network traffic and for virtualization management traffic, respectively.

We note that the emulator did not implement a node controller for these experiments. However, the original CPU demands were scaled by 1.5 to reflect a desired CPU allocation that corresponds to a utilization of allocation of 66% typical to ensure interactive responsiveness for enterprise workloads. This provided an approximation of the impact of using a node controller for this study. For scheduling, if aggregate workload demand and migration-induced CPU overhead exceeded the capacity of a node, then each workload received capacity in proportion to the number of workloads sharing the node. Any unsatisfied demands were carried forward to the next interval. Finally, migration overheads were emulated in the following way. For each workload that migrated, a CPU overhead was added to the source and the destination nodes. The overhead was proportional to the estimated transfer time based on the workload's memory size and the network interface card bandwidth. In general, we found our results to be insensitive to proportions in the range of 0.2 – 1. Therefore, we chose a factor of 0.5 of a CPU to be used throughout the transfer time.

Figure 4 shows the results of an emulation where we used the pod set controller to periodically rearrange the 138 workloads to minimize the time-varying number of active nodes. For this scenario, we assumed the pod set controller had perfect knowledge of the future and chose a workload placement such that each node was able to satisfy the peak of its aggregate workload CPU and memory demands, which gives us a theoretical baseline for comparison with algorithms that have realistic assumptions. Figure 4 shows the impact on capacity requirements of using the pod set controller once at the start of the three months (i.e., Initial Rearrangement Only) and for cases where the workload placement is recomputed every 4 Weeks, 1 Week, 1 Day, 4 Hours, 1 Hour, and 15 Minutes, respectively. The x-axis shows the Total CPU Hours used relative to the 4 Hours case. A smaller value indicates lower CPU usage. CPU Hours includes busy time and idle time on nodes that have workloads assigned to them. The cases with more frequent migrations incur greater CPU busy time, due to migration overhead, but may have lower total time if fewer nodes are needed. The figure shows that re-allocating workloads every 4 Hours captures most of the capacity savings that can be achieved. It requires 39% less CPU hours than the Initial Rearrangement Only case (1.00 vs. 1.64) and 22% less CPU hours than rearranging on a daily basis (1.00 vs. 1.28). It uses 9% and 14% more CPU hours than rearranging every hour (1.00 vs. 0.92) and 15 minutes (1.00 vs. 0.88), respectively, but it has much better CPU quality, as we discuss in the next paragraph. That is why we selected the 4 Hours case as our baseline.

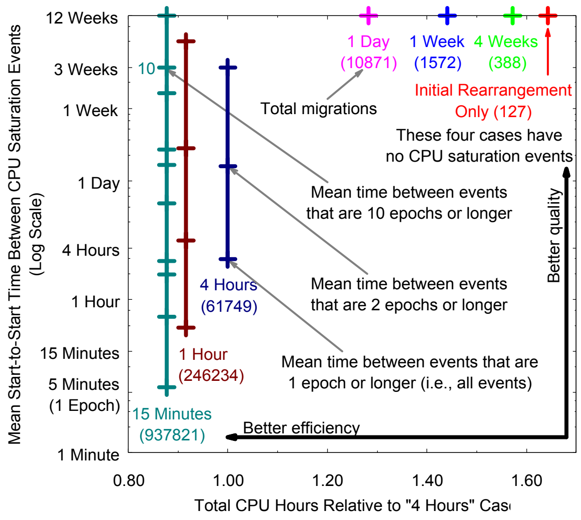
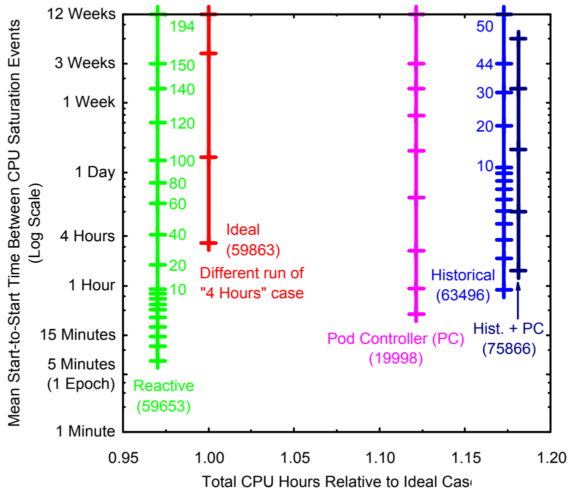
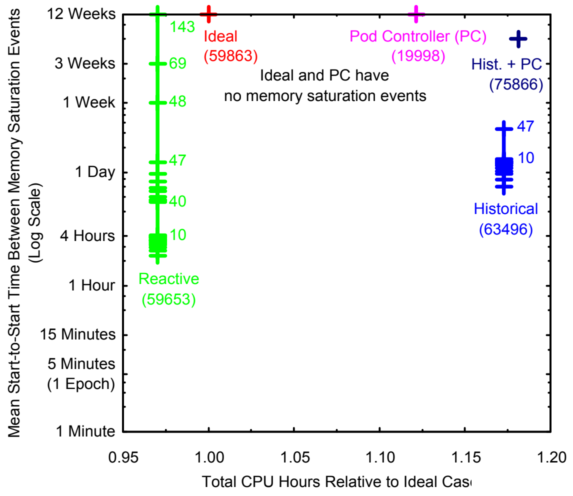


Figure 4. CPU quality vs. rearrangement periods

Even though we assume perfect knowledge of workload demands, we did not include the CPU overhead of migrations when conducting our workload placement analysis. For this reason, even the ideal cases can incur CPU saturation events. However, there was no memory overhead for migrations, so there were no memory saturation events for these cases. Figure 4 shows the frequency of CPU saturation events using a vertical bar for each case. The y-axis is a logarithmic scale for the mean period between saturation events, which is calculated by dividing 12 weeks by the number of events. The bottom tick on a bar corresponds to a saturation event of one epoch (5 minutes) or longer (i.e., all saturation events). Each tick upwards corresponds to two epochs (10 minutes) or longer, three epochs or longer, as so forth. For the 4 Hours case, there are CPU saturation events lasting five minutes or longer every three hours, ten minutes or longer every day and a half, and fifteen minutes or longer every three weeks. This is aggregated over all 138 workloads. One of the ticks in the 15 Minutes case of Figure 4 is annotated with a "10" to indicate that it corresponds with events lasting 10 epochs (50 minutes) or longer. The careful reader will observe that it is actually the ninth tick from the bottom. This is because this case has no saturation events that are 9 epochs long, so the tick for 9 epochs or longer would be in the same position as the tick for 10 epochs or longer. This notation is used more extensively in Figure 5 to reduce clutter.



(a) CPU quality vs. policy



(b) memory quality vs. policy

Figure 5. Emulation results for different pod and pod set controller policies

We now consider the impact of several different pod set and pod controller policies for managing workload migration. Figure 5 shows some results from the emulations. The Ideal case corresponds to the 4 Hours case from Figure 4 that shows an ideal balance of capacity and quality behavior. The Reactive case has a pod set controller that considers resource consumption from the previous four hours to predict the next four hours. The Historical case has a pod set controller that models the next four hours using data for a similar time period from the prior week. We expect the history based pod set controller to be beneficial, as it anticipates future demands based on past time-varying demands for a similar period. The Pod Controller (PC) case introduces the TUM pod controller. The pod set controller chooses an initial workload placement but subsequently the TUM pod controller works with 5 minute data to initiate workload migrations and shutdown and startup nodes as required. Finally, the Historical + PC case uses an integration of both the TUM pod controller and the Historical pod set controller.

The use of a Pod Controller (PC) alone is most typical of the literature [12]. Figure 5 shows that the use of a pod set controller in a Reactive mode performs worse than the PC case in terms of quality. It systematically underestimates capacity requirements. It uses 13% and 3% less CPU hours than the PC case (0.97 vs. 1.12) and the Ideal case (0.97 vs. 1.00), respectively. However, it incurs CPU saturation events with high frequency and length. In particular, the Reactive case has longer saturation events occurring more frequently than correspondingly long events in the PC case. From detailed results, the Reactive case repeatedly underestimates the capacity needed for backups just past midnight each day. The Historical pod set controller overcomes this issue, but it uses 4% more CPU hours than the PC case (1.17 vs. 1.12) and still incurs lower CPU quality than PC alone. We note that the Historical case does not react to differences between predicted and actual demands. By integrating the Historical pod set controller and the Pod Controller (PC), we use 5% more CPU hours than the PC case (1.18 vs. 1.12), although this case does use 28% less CPU hours than the Initial Rearrangement Only case above (1.18 vs. 1.64). Also, it has significantly improved CPU quality over all other cases in Figure 5, excluding the theoretical Ideal case. The frequency of CPU saturation events is lower and there are fewer long events. The integrated controllers benefit from predicting future behavior, as well as reacting to current behavior in a timely manner. Figure 5(b) shows that Historical + PC case incurs a five minute memory saturation event every six weeks on average, where memory demand exceeds supply on a node. These are incurred after the pod set controller rearranges workloads but before the pod controller reacts to memory overload.

4.2. Integrated pod and node controller

Another case study was done on our experimental testbed described in Section 3.2 to validate the effectiveness of the integration between the node controller and CF pod controller. We ran 16 Apache Web servers in 16 Xen VMs on 4 physical nodes in a pod. The workloads were driven using CPU consumption traces from two Web servers, 10 e-commerce servers and 2 SAP application servers from various enterprise sites. The workloads are associated with two classes of service, where eight of them belong to the High Priority- (HP-) class and the other eight belong to the Low Priority- (LP-) class. We start with a semi-random initial placement of workloads, where each node hosts four workloads, two in the HP-class and two in the LP-class. We consider a resource utilization target of 70% and 80% for HP-class and LP-class workloads, respectively, to provide service level differentiation between the two classes. During resource contention on a node, the resource requests of HP-class workloads are satisfied, if possible, before the workloads in the LP-class get their shares.

We compare three workload management solutions in this experimental study:

- **Fixed Allocation** (“no control”): Each VM (incl. Dom-0) has a fixed 20% allocation of its node’s CPU capacity. There are no VM migrations.
- **Independent control**: The CF pod and node controllers run in parallel without integration.
- **Integrated control**: The CF pod and node controllers run together with integration.

The first solution is one without dynamic resource control, and it simply provides a baseline for the study. The control intervals for the pod and node controllers are 1 minute and 10 seconds, respectively.

Figure 6 shows a comparison of the resulting application performance from using the three solutions. From the client side, a mean response time (MRT) is computed and logged every 10 seconds over the duration of each experiment (5 hours). To better illustrate the results, we consider a 2-sec MRT target for the HP-class workloads and a 10-sec target for the LP-class workloads. A cumulative distribution of the MRT across all 16 workloads for the Fixed Allocation case is shown with the dashed green line. No class of service was considered in this solution. All the workloads achieve the 2-sec target 68% of time and the 10-sec target 90% of time. The two blue lines represent the MRT distributions for the independent control solution. More specifically, the solid blue line and the dashed blue line correspond to the HP-class and the LP-class workloads, respectively. As we can see, the HP-class workloads achieve the 2-sec target 73% of time (a 5% improvement over Fixed Allocation), but the LP-class workloads achieve the 10-sec target only 67% of time. With the integrated control solution, the MRT distributions for the HP-class and the LP-class workloads are represented by the solid red line and the dashed red line, respectively. We see that the HP-class workloads achieve the 2-sec target 90% of time, an improvement of 22% and 17% over the no control and independent control solutions, respectively. The relative improvements in these two cases are 32% (22/68) and 23% (17/73), respectively. The LP-class workloads achieve the 10-sec target 70% of time, similar to the no integration case.

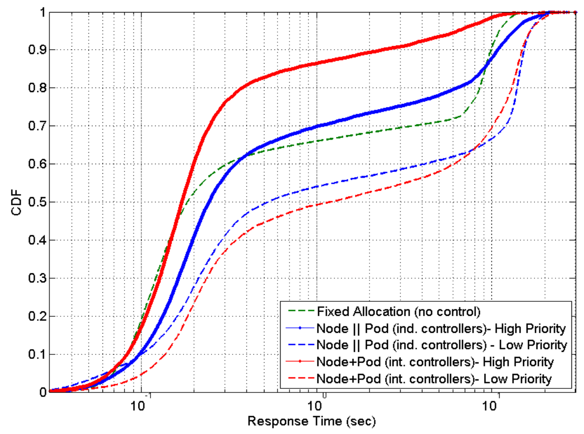


Figure 6. Cumulative distributions of 10-second mean response times for all the workloads from using three workload management solutions - no control (green), independent control (blue), and integrated control (red)

Table 1. Comparison of migration events and unsatisfied demand with and without integration

	No. of migration events		Unsatisfied demand (% of total demand)	
	HP	LP	HP	LP
Independent	17	14	15	12
Integrated	13	22	9	15

To explain the observed difference between the two controller solutions, we recall that without integration, the pod controller estimates workload resource demand based on the observed resource consumptions only. In contrast, when the two controllers are integrated, the node controller determines the resource allocation each workload needs to satisfy its performance goal, and this information is provided to the pod controller as an input. The results in Figure 6 clearly show that this integration enables the pod controller to take into account the performance-driven resource demands of all the workloads, and therefore make better workload placement decisions such that the HP-class workloads have higher probabilities of achieving their service level objectives.

In addition, we computed the statistics of system-level metrics from the controller logs to see if they demonstrate similar trends as seen in the response time data. Table 1 shows a comparison of the two controller solutions in terms of two metrics: the total number of VM migrations that occurred and the total unsatisfied demand (resource request) as a percentage of total demand, for both the HP-class and LP-class workloads. As we can see, the HP-class workloads experienced a higher number of migrations using independent control (17) than using integrated control (13). This is consistent with our previous explanation that when resource requests are considered instead of measured consumptions, the HP-class workloads are less likely to be migrated and consolidated, leading to better performance. Similarly, the integrated control solution resulted in a lower percentage of unsatisfied demand (9%) compared to the independent control solution (15%). Both statistics are consistent with the observed response time data shown in Figure 6.

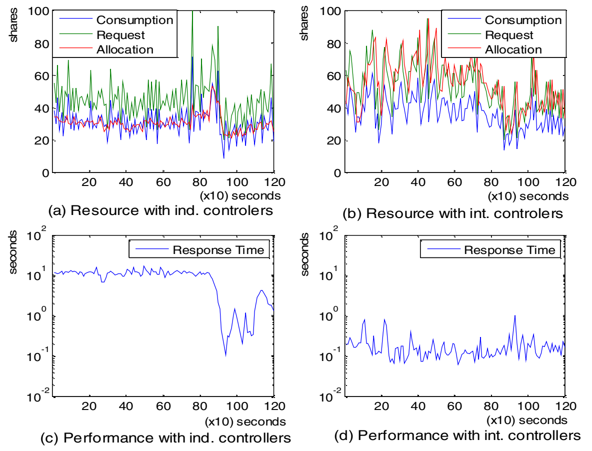


Figure 7. Time series of the resource consumption (blue), request (green), and allocation (red), as well as measured mean response time (bottom) for an HP-class workload, with independent control in (a) and (c), and integrated control in (b) and (d).

In Figure 7, we demonstrate the impact of controller integration on a particular HP-class workload. The two top figures show the measured CPU consumption, the CPU request computed by the utilization controller, and the actual CPU allocation determined by the arbiter controller for this workload over a 20 minutes interval. The two bottom figures show the resulting response times. The two figures on the left represent the independent control case. As we can see, the actual allocation is below the request most of the time, causing the VM hosting this workload to be overloaded (in 7(a)), resulting in high response time for most of the 20 minutes interval (in 7(c)). Note that the response time drops off at around 900 seconds, which is due to the reduced resource demand as we can see from 7(a). In a few sampling intervals that follow, the allocation is below the request but above the average consumption, which means that the VM is less overloaded and has a much lower response time. The two figures on the right show the result for the integrated control solution, where the CPU request is mostly satisfied (in 7(b)), leading to much lower response time (in 7(d)). The advantage of the integrated control solution is that it made more informed placement decisions that did not subject this HP-class workload to an overload situation.

5. Related work

VMware's VirtualCenter and DRS products [2] and the management infrastructure from Virtual Iron [4] provide alternatives to parts of our solution. Each offers a degree of pod control for workloads in hypervisor-based VMs. Our approach considers additional metrics, like application service level metrics (e.g., response time and throughput), and uses long-term historical usage trends to predict future capacity requirements. The commercial products could possibly be integrated into our architecture.

The work in [13] integrates various sophisticated aspects of power and performance management at the node and the pod levels. It presents a simulation study that optimizes with respect to power while minimizing the impact on performance. The simulation results for integrated control suggest that between 3% and 5% of workload CPU demand is not satisfied, but unsatisfied demands were not carried forward between simulation periods. Our host emulation approach carries forward demands and focuses more on the length of events where performance may be impacted

Khana *et al.* solves the dynamic VM migration problem using a heuristic bin-packing algorithm, evaluated on a VMware ESX Server testbed [12]. Wood *et al.* consider black and grey box approaches for managing VM migration using a combination of node and pod controller in a measurement testbed [14]. They only consider resource utilization for the black box approach, and add OS and application log information for the grey box approach. They find that the additional information helps to make more effective migration decisions. Neither work takes advantage of long-term demand patterns as we do using the pod set controller.

Control theory has recently been applied to performance management in computer systems [15] through admission control [16][17] or resource allocation [18][19][20], including dynamic resource allocation in virtualized environments [21][22][5]. Compared with these prior solutions that only dealt with individual non-virtualized or virtualized systems, we have proposed an integrated solution for capacity and workload management in a virtualized data center through a combination of dynamic resource allocation, VM migration, and capacity planning.

6. Conclusion and future work

In this paper, we introduce the 1000 Islands solution architecture that integrates islands of automation to the benefit of their managed workloads, as well as our first steps toward an implementation of this architecture.

While all of the controllers achieve their goals independently using different analytic techniques, including control theory, meta-heuristics, fuzzy logic, trace-based analysis, and other optimization methods, there is power in leveraging each controller independently and then combining them in this unified architecture. In the emulations, the integrated pod set and pod controllers resulted in CPU and memory quality that approached that of the hypothetical ideal case, while using only 18% more capacity. The testbed showed that the integration of pod and node controllers resulted in performance improvements of 32% over the fixed allocation case and 23% over the non-integrated controllers, as well as reduced migrations for high priority workloads. In addition, service level differentiation can be achieved between workload classes with different priorities.

As a next step, we plan to scale our testbed to a larger number of physical nodes so that they can be divided into multiple pods. This will allow us to evaluate the complete solution architecture that consists of node, pod, and pod set controllers on real systems, as well as study consolidation scenarios with a much larger number of workloads.

We will also integrate with power [13] and cooling [23] controllers, to better share policies and to offer a more unified solution for managing both IT and facility resources in a data center. For example, our node controller can be extended to dynamically tune individual processor P-states to save average power, our pod controller can consider server-level power budgets, and the thermal profile of the data center can guide our pod set controller to place workloads in areas of lower temperature or higher cooling capacity.

Ultimately, data center operators would like application-level service level objectives (SLOs) to be met without having to worry about system-level details. In [24], workload demands are partitioned across a two priorities to enable workload-specific quality of service requirements during capacity planning and runtime phases. This can be integrated with node and pod controllers. In [25], application-level SLOs are decomposed into system-level thresholds using performance models for various components being monitored. These thresholds can potentially be used to drive our utilization controllers at the VM level. However, this decomposition is done over longer time scales (minutes). In [5], we have developed a feedback controller for translating SLO-based response time targets into resource utilization targets over shorter time scales (seconds). These approaches can be incorporated in our next round of integration. Finally, a distributed management framework is being developed for integrating all of these components in a scalable manner, such that they can potentially manage a data center of 10,000 nodes.

7. References

[1] HP Virtual Connect Enterprise Manager: http://h18004.www1.hp.com/products/blades/components/ethernet/vcem/index.html

[2] VMware ESX Server: http://vmware.com/products/vi/esx/

[3] Citrix XenServer: http://www.citrixxenserver.com/products/Pages/XenEnterprise.aspx

[4] Virtual Iron: http://www.virtualiron.com/products/

[5] X. Zhu, Z. Wang, and S. Singhal, "Utility-Driven workload management using nested control design," *American Control Conference (ACC'06),* June, 2006.

[6] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, "Live migration of virtual machines," *the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05),* May 2005.

[7] C. Hyser, B. Mckee, R. Gardner, and B.J. Watson, "Autonomic virtual machine placement in the data center," *HP Labs Technical Report HPL-2007-189*, 2007.

[8] S. Seltzsam, D. Gmach, S. Krompass and A. Kemper. "AutoGlobe: An automatic administration concept for service-oriented database applications," *the 22nd Intl. Conference on Data Engineering (ICDE'06), Industrial Track*, 2006.

[9] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi and A. Kemper, "An integrated approach to resource pool management: policies, efficiency and quality metrics," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08),* 2008.

[10] J. Rolia, L. Cherkasova, M. Arlitt and A. Andrzejak. "A Capacity Management Service for Resource Pools," *the 5th Intl. Workshop on Software and Performance (WOSP'05)*, Spain, 2005.

[11] D. Mosberger and T. Jin, "Httperf—A tool for measuring Web server performance," *Workshop on Internet Server Performance*, 1998.

[12] G. Khana, K. Beaty, G. Kar and A. Kochut, "Application Performance Management in Virtualized Server Environments," *IEEE/IFIP Network Operations & Management Symposium (NOMS),* April, 2006.

[13] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No power struggles: Coordinated multi-level power management for the data center," *ASPLOS 2008*.

[14] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," *the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI '07),* April, 2007.

[15] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*, ser. ISBN: 0-471266-37-X. Wiley-IEEE Press, August 2004.

[16] A. Kamra, V. Misra, and E. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," *International Workshop on Quality of Service (IWQoS)*, 2004.

[17] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance isolation and differentiation for storage systems," *International Workshop on Quality of Service (IWQoS)*, 2004.

[18] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, 2002.

[19] Y. Lu, T. Abdelzaher, and A. Saxena, "Design, implementation, and evaluation of differentiated caching services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, May 2004.

[20] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," *International Conference on Autonomic Computing (ICAC'05)*, 2005.

[21] Z. Wang, X. Zhu, and S. Singhal, "Utilization and SLO-based control for dynamic sizing of resource partitions," *the 16th IFIP/IEEE Distributed Systems: Operations and Management (DSOM'05),* October, 2005.

[22] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. Shin, "Adaptive control of virtualized resources in utility computing environments," *EuroSys2007*, March 2007.

[23] C.E. Bash, C.D. Patel, R.K. Sharma, "Dynamic Thermal Management of Air Cooled Data Centers", *Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, 2006.

[24] L. Cherkasova and J. Rolia, "R-Opus: A composite framework for application performability and QoS in shared resource pools," *International Conference on Dependable Systems and Networks (DSN'06),* 2006.

[25] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, "SLA decomposition: Translating service level objectives to system level thresholds," *4th IEEE International Conference on Autonomic Computing (ICAC'07)*, June 2007.