Adaptive Live Migration to Improve Load Balancing in Virtual Machine Environment

Peng Lu, Antonio Barbalace, Roberto Palmieri, and Binoy Ravindran

ECE Department, Virginia Tech, USA
{lvpeng,antoniob,robertop,binoy}@vt.edu

Abstract. Load balancing is one of the main challenges in a Virtual Machine (VM) Environment in order to ensure equal utilization of all the available resources while avoiding overloading a subset of machines. In this paper, we propose an efficient load balancing strategy based on VM live migration. Unlike previous work, our strategy records the history of mappings to inform future placement decisions. We also apply a workload-adaptive live migration algorithm in order to minimize VM downtime and improve the user experience. The evaluation shows that our load balancing technique is faster than previous approaches, thus reducing the decision generating latency by as much as 79%. Furthermore, the results also show that we provide minimal downtime. Compared with competitors, our proposed migration mechanism reduces the downtime by up to 73%.

1 Introduction

Virtual machine (VM) environments, where multiple physical machines and VMs are interconnected via increasingly fast networking technologies, are becoming extremely popular for providing on-demand computational services and high availability. One of the most interesting features is that the pool of available resources (CPU, virtual memory, secondary storage, etc.) can be shared between all the machines through the available interconnections. Several previous works [10,13,17] observed that many resources are actually unused for a considerable amount of the operational time. Therefore, inter-machine load balancing is of great interest in order to avoid situations where some machines are overloaded while others are idle or under-used.

Traditionally, there are several ways to achieve load balancing in networked systems. One straightforward solution is static load balancing, which assigns applications at the beginning. The efficiency of this strategy depends on the accuracy of the prior prediction. A dynamic load balancing strategy can be exploited by migrating processes at run time among different machines in a networked environment, rather than limiting the applications to run on the machine where they are first assigned. To overcome the limitations imposed by process migration, traditionally used in cluster environments, an OS-level methodology that migrates all the running applications together with their hosting OS was presented. This methodology attracted more attention than process migration,

especially through a technique called VM live migration [9]. Live migration is migrating a VM from the source Virtual Machine Monitor (VMM) to the target VMM (usually running on a different host machine) without halting the migrating guest operating system. Most state-of-the-art/practice VMMs (e.g., Xen [7], VMware [22], KVM [1]) provide live migration mechanisms today. Unlike process migration, VM live migration transfers the virtual CPU and emulated devices' state and the guest OS memory data all together. Live migration is the enabling technology behind many beneficial features such as load balancing, transparent mobility, and high availability services. In addition, because it relies on VMM to make a clean separation between hardware and software, live migration also aids in aspects such as fault tolerance, application concurrency, and consolidation management.

VM live migration has several advantages regarding the load balancing problem. A VM hosting several different running applications on a heavily loaded machine can be migrated to another VMM laying idle on another machine in order to exploit the availability of all the resources. Also, adopting live migration instead of a stop and resume migration mechanism natively ensures minimal downtime and it minimally interrupts the VM user's interaction. Here, the downtime is defined as the waiting time between when the VM is unusable by the users (stopped on the source VMM) and when the VM is working again (resumed on the target VMM). We believe a good load balancing strategy should provide minimal downtime for users.
This paper makes the following contributions:

1. We propose a fast and efficient centralized load balancing framework based on history records. The evaluation shows that our load balancing technique is an effective method, while it also reduces the decision generating latency as much as 79% compared with previous approaches.

2. We apply a workload-adaptive approach to provide minimal downtime for different kinds of applications. Results show that our proposed migration mechanism reduces the downtime by up to 73% compared to competitors.

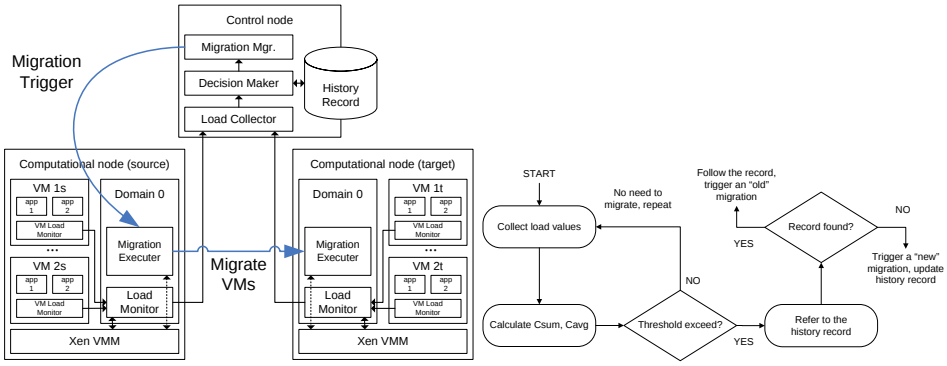
The rest of the paper is organized as follows: Section 2 discusses past and related work. Section 3 presents the design and implementation of our load balancing and live migration mechanism. Section 4 reports our experimental environment, benchmarks used, and evaluation results. We conclude and discuss future work in Section 5.

2 Related Work

The general problem of load balancing in a networked environment has been examined for decades using different strategies [8,11,21]. In user space, Ansel *et. al.* [6] allow dynamic migration of processes by checkpointing and restart. Heo *et. al.* [24] rely on the cooperation between the process and the migration subsystem to achieve its migration. The problem in these user space implementations is that without kernel access, they are unable to migrate processes with location dependent information and interprocess communication.In kernel space, imple-

mentations like [14, 23] allow the migration process to be done more quickly and are able to migrate processes with dependencies. Compared with user space process migration, they provide better performance. Although kernel space process migration techniques are more efficient, they require modifications to the operating system kernel.

Some research has been proposed to improve the load balancing performance using VM live migration. Different works [12, 15, 16, 18, 20, 25] use a prediction method to forecast future resource demands based on the current resource utilization. However, in order to get an accurate prediction, these works need to obtain and analyze the system performance continuously, which introduces overhead. Our load balancing strategy doesn't make predictions in advance; instead, we always update and refer to the history record to assist in generating the final decision. Furthermore, previous work didn't consider the migration cost, and the downtime performance is not evaluated in their experimental results. One of our contributions is to design an adaptive VM migration mechanism in order to provide minimal downtime for different kinds of applications. Additionally, we characterize and quantify the downtime, and present the results in Section 4.2.



(a) The architecture of DCbalance. (b) Execution flow of DCbalance.

Fig. 1: DCbalance framenwork.

3 Design and Implementation

3.1 Load Balancing Algorithm

Our prototype includes several machines which are linked by a network (Figure 1a). We choose one machine as the control node running only the load balancing strategy. The other machines are used as computational nodes with several VMs running on top of them. The load balancing mechanism is centralized: the control node runs the cloud management software that controls all the nodes running VMs.

From a system-level point of view, when a node is heavily-loaded while others are lightly-loaded, the imbalance problem emerges and it's time to generate a load balancing decision. The dynamic load balancing strategy continuously attempts to reduce the differences among all the nodes, by migrating VMs from heavily-loaded to lightly-loaded nodes. Figure 1b breaks down our strategy into steps, which are explained below.

1. **Collect the load values on each computational node.** In this paper we focus on CPU load-balancing and memory load-balancing of different nodes in a VM environment. We model each node as a gray box, which provides different interfaces to live record CPU and memory load information for each node and each running virtual machine.
   We use two methods to measure the load value at user level in Linux: by either using the Linux syscall `sysinfo()`, or reading the information residing in the `proc` file system. For CPU information we check the total `jiffies` that the processor spent executing each VM and the physical machine as well, and compute the utilization percentage for each part. For memory information we record the page operations per second, for example, the number of page faults requiring a page to be loaded into memory from disk.
2. **Determine whether to trigger the live migration or not.** After the control node collects the load values from all computational nodes, the total and average utilizations of all the machines involved in load balancing are computed. We adopt a threshold based strategy for deciding when virtual machines should be migrated between the nodes. The VM live migration will be triggered if C_m, the mean value of the sum of the maximum and minimum distances (respectively $C_{diff_{max}}$ and $C_{diff_{min}}$) from the average utilization (C_{avg}), is greater than a threshold T. C_i is the total CPU utilization of the i^{th} computational node, $i \in 1, .., n$; n is the current number of nodes in the VM environment.
   $C_{avg} = \frac{1}{n} \sum_{i=1}^{n} C_i$
   $C_{diff_i} = C_i - C_{avg}$
   $C_{diff_{max}} = \max_{i \in \{1,..,n\}} C_{diff_i}$
   $C_{diff_{min}} = \min_{i \in \{1,..,n\}} C_{diff_i}$
   $C_m = (C_{diff_{max}} + C_{diff_{min}})/2$
3. **Schedule the live migration by checking the load balancing history record.** Whenever a migration is triggered, the control node checks the history record for a similar CPUs utilization scenario, i.e. a similar load distribution on the computational nodes. Note that the same C_{avg} doesn't ensure the same CPU utilizations scenario; the mapping is the key. If a previous record exists, we can schedule the VM live migration by choosing the same source and destination nodes. If several similar records exist, we just follow the latest record and schedule the migration. If we can't find such a record, the current situation is totally new and the nodes with C_i value close to $C_{diff_{max}}$ are used as sources, while the nodes with C_i value close to $C_{diff_{min}}$ are used as destinations of the live migration. After the migration, we add such situation (or mapping) as a new entry in the history record.

3.2 Workload Adaptive Live Migration

As opposed to previous works, we aim to provide load balancing with minimal downtime via virtual machine live migration. We focus on effectively balancing the usage of two kinds of resources: the CPU and memory. In our preliminary experiments, however, we found that a general VM live migration mechanism doesn't work well in all cases, such as when dealing with a memory-intensive application. Therefore, we implement a workload-adaptive migration mechanism.

- **Generic application live migration.** By "generic" we mean that there is no memory-intensive workload running in the guest VM, but rather a mixed workload, like Apache [4]. In this case the live migration is implemented by exploiting the pre-copy technique [9, 19], however, we compress the dirty memory data before transmitting it. Compressed data takes less time to be transferred through the network. In addition, network traffic due to migration is significantly reduced when less data is transferred between different computational nodes. The technique is summarized in the following.
  1. After the migration is triggered, all the memory pages of the selected VM are transmitted from the source node to the target node while the VM is still running (first migration epoch).
  2. For subsequent migration epochs, the mechanism checks the dirty bitmap to determine which memory pages have been updated during the epoch. Only the newly updated pages are transmitted. The VM continues to run on the source node during these epochs.
  3. Before transmitting in every epoch, the presence of dirty data is checked in an address-indexed cache of previously transmitted pages. If there is a cache hit, the whole page (including this memory block) is XORed with the previous version, and the differences are Run-Length Encoded (RLE). Only the differences (*delta*) from a previous transmission of the same memory data are transmitted.
  4. For the memory data which is not present in the cache we apply a general-purpose quick compression technique implemented in zlib [5].
  5. When this mechanism is no longer beneficial, the VM is stopped on the source node, the remaining data (left pages, CPU registers and device states, etc.) is transmitted to the target node, and the VM is resumed there.
- **Memory-intensive applications live migration.** When dealing with memory-intensive applications in which memory is updated at high frequency, the above presented live migration doesn't work well because it leads to unacceptable overheads. Assume a memory page is frequently updated by a memory-intensive application; its updated copy will be compressed and transferred on each migration epoch. It could be worse considering that a VM can be assigned up to several gigabytes of memory to run the guest OS. Therefore, to load balance workloads characterized by memory-intensive applications, we propose another live migration technique described in the following.

1. After the migration is triggered, all the memory pages of the selected VM are compressed and transmitted from source to the target node while the VM is still active.
2. The VM is then suspended on the source node until a minimal and necessary execution state (or checkpoint) of the VM (including CPU state, registers, and some non-pageable data structures in memory) has been transmitted to the target and resumed there.
3. On the target node, if the resumed VM tries to access a memory page that has not been updated, a page fault will be generated and redirected towards the source. The source node will respond to the fault by fetching the corresponding memory pages, then compressing and transmitting them to the target node (we adopt the same compression method described above).

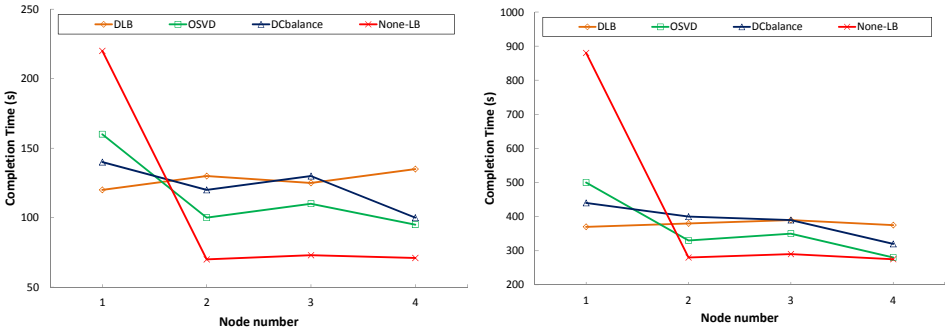
When the applications running in the VM are mostly memory-intensive, this design ensures that each memory page is compressed and transmitted at maximum twice during a migration. Additionally, if the workload is read-intensive most memory data is only transmitted once.

4 Experimental Evaluation

Our experimental environment includes five machines: one is used as the control node, and the other four (computational nodes) are running the guest VMs. Each computational node is configured with an Intel i5 processor running at 2.3GHz with either 4GB or 6GB of RAM. All machines are connected by a 1Gb Ethernet switch. Our load balancing strategy and migration mechanisms are based on Xen(3.4.0). The operating system (installed on each of the five machines and guest VMs) is Centos 5.2 with the Linux 2.6.18.8 kernel. We refer to our load balancing framework as *DCbalance*. Our competitors include OSVD [16] and DLB [15]. OSVD is also based on VM live migration and integrates performance prediction mechanism. DLB implements a dynamic load balancing algorithm [15] deployed into Xen's original live migration mechanism. We split the performance evaluation in two: a first section about our load balancing strategy and another section about the proposed migration mechanism.

4.1 Load Balancing Strategies Comparison

For this evaluation we run the MPI version of the NPB test suite [2]. We present here the results about the embarrassing parallel (EP) test. EP is a compute-bound benchmark with few network communications [2]. Initially we let each computational node run the same number of VMs. The total workload is divided and assigned to all the VMs on the 4 nodes. We let *node1* execute 50% of the total workload while the remaining 3 nodes execute the rest (16.6% each). Obviously *node1* is overloaded. Each node runs 8 VMs with 256MB assigned as guest memory. Initially the workload assigned to each node is distributed to the VMs



(a) Running NPB-EP.C program (b) Running NPB-EP.D program

Fig. 2: Workload completion time comparison on different problem data sizes.

running on the that node. In this experiment we set the threshold T to 0.1 (see Section 3.1).

We first compare the three load balancing strategies with the case that no load balancing strategy is used. Figure 2a shows the completion times when running the class C problem size of the NPB-EP benchmark, "none-LB" is the case in which no load balancing strategy is applied to the workload. The x-axis refers to the node number while the y-axis is the workload completion time in seconds. As mentioned above, initially each node has 8 VMs on average and *node1* is overloaded with half of the total workload. Without any load balancing strategy, *node1* took close to 220 seconds to complete the workload while all other nodes took only about 70 seconds each. When applying any of the three load balancing strategies, however, all the nodes exhibit relatively consistent completion times because the VM(s) on the overloaded node were already migrated to other nodes. We observe that compared with other published load balancing algorithms, DCbalance achieves similar performance. Figure 2b shows the same evaluation but on a larger dataset. Our load balancing strategy can still provide comparable performance when dealing with the larger problem.

Table 1: Workload percentage after migration and triggering latency comparison.

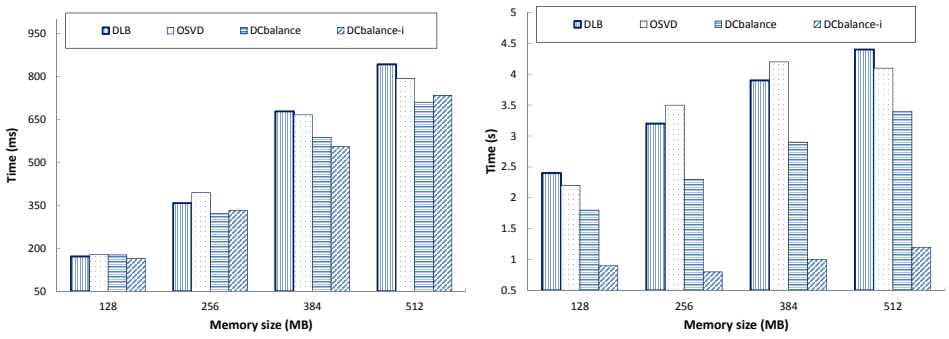
LB strategy	#1 Load	#2 Load	#3 Load	#4 Load	Triggering latency
DLB	24%	25%	26%	25%	2.2s
OSVD	27%	35%	17%	21%	3.1s
DCbalance	28%	26%	23%	21%	0.63s
DCbalance-nh	29%	25%	27%	19%	1.09s

We also compare the triggering latency of each load balancing strategy; i.e. the elapsed time between the collection of load information and the time when

the decision is generated. We compute the workload percentage as well as the VMs load values on each node after the migration finish. In this test we used a history record of (the most recent) 20 entries. To evaluate the benefits of the history record, we add another competitor: *DCbalance-nh*, which is our load balancing design without any history record (we set the number of entries to zero). Table 1 shows the results under four load balancing strategies. From the numbers we observe that the DLB algorithm outperforms the others but it needs longer time to generate the decision. The OSVD system incurs even more time due to prediction overhead, and we can also observe that its final decision is not as balanced as DLB. Compared with these two strategies, the main benefit of our strategy is the short time needed for finding the solution, which reduce the triggering latency by as much as 79% compared with the OSVD system. We also observe that it is an efficient method, as the migration decision is comparably fair. From the differences between DCbalance and DCbalance-nh, we conclude that by referring to the history record we gain performance improvements by more quickly generate the migration decision.

4.2 Migration Mechanisms Comparison

We evaluate the performance of migrating VMs running two types of workloads: "generic", by runing the Apache [4] webserver and "memory-intensive", by running Sysbench [3] (e.g., 25% read operations and 75% write operations). We let each node run 4 VMs with the same assigned memory size. We conduct the evaluation while varying the guest memory size from 128MB to 512MB, in order to investigate the impact of memory size on the downtime and other performance characteristics. To verify the effectiveness of our adaptive migration mechanism we again add another competitor, *DCbalance-i*. We refer to our migration design for generic applications as DCbalance, whereas we refer to its improved version which better handles memory-intensive applications as DCbalance-i.



(a) Downtime under Apache (b) Downtime under Sysbench

Fig. 3: Downtime comparison under different workloads.

Figure 3a shows the downtime results for the Apache benchmark for four migration mechanisms with three different sizes of assigned guest memory. We observe that all four mechanisms incur minimal downtime (within 1 second). However, because our proposed mechanism further reduces the data to be transmitted by compressing it before migration, it leads to lower downtime numbers. As the assigned guest memory goes up, the gap becomes larger, with a 17% reduction under the 512MB case. Another observation is that there is no obvious difference between the results of DCbalance and DCbalance-i. This is because the DCbalance-i is designed for memory-intensive applications but the Apache benchmark is not memory bounded.

Figure 3b shows the downtime results in the same cases as Figure 3a while running a memory-intensive workload (Sysbench benchmark) which updates the guest memory at high frequency. The findings are clearly different from the previous case. DCbalance still performs better than DLB and OSVD, which incur several seconds each; it reduces the downtime by roughly 35%. However, the DCbalance-i is by far the best mechanism, as it maintains a downtime around 1s in all cases. The downtime is reduced by 73% compared with DLB and OSVD.

5 Conclusion

Our strategy is proven to be fast and efficient as a load balancing approach. Instead of making predictions in advance, we refer to a history record to help schedule the VM migration. Decision generation is up to 79% faster than competitors. Moreover, we apply a workload-adaptive migration mechanism to provide minimal downtime for different kinds of applications, so as to improve the user experience. Results show that our proposed migration mechanism reduces the downtime by up to 73% in comparison to competitors. Future work includes combining the network topology and network communication monitoring with our proposed method in order to make network-aware mapping decisions (and thus reduce the network traffic). Furthermore, we will extend the problem of load balancing to distributed VM environments.

References

1. Kvm: Kernel based virtual machine. `www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`.
2. NASA NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.
3. Sysbench benchmark. `http://sysbench.sourceforge.net`.
4. The Apache HTTP Server Project. `http://httpd.apache.org/`.
5. Zlib memory compression library. `http://www.zlib.net`.
6. J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. 2009.
7. 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. SOSP, 2003.

8. E. Caron, F. Desprez, and A. Muresan. Forecasting for grid and cloud computing on-demand resources based on pattern matching. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010.
9. C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. NSDI, 2005.
10. C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, 2011.
11. C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. pages 313–324, 2011.
12. J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Proceedings of the 11th IFIP/IEEE INM*, IM'09, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press.
13. M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 832–843, Washington, DC, USA, 2009.
14. J. Hursey, T. I. Mattox, and A. Lumsdaine. Interconnect agnostic checkpoint/restart in open mpi. Technical report, New York, NY, USA, 2009.
15. P. Jain and D. Gupta. An algorithm for dynamic load balancing in distributed systems with multiple supporting nodes by exploiting the interrupt service. *International Journal of Recent Trends in Engineering*, 1(1):232–236, 2009.
16. Z. Li, W. Luo, X. Lu, and J. Yin. A live migration strategy for virtual machine based on performance predicting. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 72 –76, aug. 2012.
17. C. Liu, B. T. Loo, and Y. Mao. Declarative automated cloud resource orchestration. pages 26:1–26:8, 2011.
18. P. Lu, B. Ravindran, and C. Kim. Enhancing the performance of high availability lightweight live migration. In *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin Heidelberg, 2011.
19. M. Nelson, B. H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05*, pages 25–25, Berkeley, CA, USA, 2005.
20. Y. Song, Y. Li, H. Wang, Y. Zhang, B. Feng, H. Zang, and Y. Sun. A service-oriented priority-based resource scheduling scheme for virtualized utility computing. HiPC, 2008.
21. R. Subrata, A. Y. Zomaya, and B. Landfeldt. Cooperative power-aware scheduling in grid computing environments. volume 70, pages 84–91, Orlando, FL, USA, Feb. 2010. Academic Press, Inc.
22. C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.
23. J. Walters and V. Chaudhary. Replication-based fault tolerance for mpi applications. *Parallel and Distributed Systems, IEEE Transactions on*, 2009.
24. C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 43:1–43:12, Piscataway, NJ, USA, 2008.
25. J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, Sept. 2008.