An Analysis of MOSIX Load Balancing Capabilities

Siavash Ghiasvand, Ehsan Mousavi Khaneghah, Sina Mahmoodi Khorandi, Seyedeh Leili Mirtaheri, Najmeh Osouli Nezhad, Meisam Mohammadkhani, and Mohsen Sharifi
School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
ghiyasvand@comp.iust.ac.ir, emousav@iust.ac.ir, sina_mahmoodi@comp.iust.ac.ir, Mirtaheri@iust.ac.ir,
{n_osouli, m_mohammadkhani}@comp.iust.ac.ir, and msharifi@iust.ac.ir

***Abstract*—Mosix has long been recognized as a distributed operating system leader in the high performance computing community. In this paper, we analyze the load-balancing capabilities of a Mosix cluster in handling requests for different types of resources through real experiments on a Mosix cluster comprising of heterogeneous machines.**

Keywords-Mosix Load balancing; CPU–intensive process; I/O-intensive process; IPC-intensive process; Memory- intensive process.

I. Introduction

Mosix used to be a pioneerd distributed operating system for cluster computing. It was built as an extension to the UNIX operating system kernel and provided a single system image to applications. Using Mosix, developers could build SMP machines by clustering a number of dispersed homogeneous machines running under UNIX.

Load balancing the computational power of clustered machines is a well-known mandatory requirement to the provision of single system image and performance (i.e., response time) in homogeneous clusters [1]. Mosix has used a decentralized and probabilistic approach to balance the computational loads on clustered off-the-shelf machines. It has used a special mechanism for scalable dissemination of load information too. To prevent process or system thrashing, it has also used a process reassignment policy through sharing of memory.

In this paper, we intend to show experimentally the capabilities of Mosix in balancing the loads on a Mosix cluster comprised of heterogeneous machines. We have designed and ran a number of tests to investigate if Mosix achieves its acclaimed capabilities in balancing the load on the cluster when requests for different cluster resources are called.

The rest of paper is organized as follows. Section 2 presents a brief description of the Mosix load-balancing mechanism. Section 3 presents our test programs and the results of running them on a 5-node Mosix cluster, and Section 4 concludes the paper.

II. Mosix load balancing mechanism

The load balancing mechanism of Mosix works in a decentralized manner using load information of the clustered machines. Process is the basic unit of load balancing in Mosix . It includes appropriate mechanisms such as process migration and remote system call for adaptive load balancing. It reassigns processes when it decides to balance the load between two nodes [2].

Mosix load balancing mechanism is based on three components, load information dissemination, process migration, and memory information dissemination. Mosix can assign a new process to the best machine as well as balancing the load on entire cluster [3]. In the remaining parts, we describe load dissemination and balancing mechanisms and our experiments written in C language based on process behaviors.

Mosix employs CPU load, memory usage, and IPC information to calculate machine's load and process needs. Indeed these are load indices in Mosix. Mosix monitors these resources, gathers their information and status, and packs them into a message [3], [4]. It then sends the built message to some other machines. Mosix uses depreciation to calculate load information. When a new index is calculated, Mosix changes it with respect to the depreciation rate. With this idea, Mosix employs a history-based approach to balance the load. Mosix monitors resources many times in a second. At the end of each second, Mosix normalizes indices and sends them into two randomly selected machines in the cluster. Mosix stores information about a few numbers of machines due to scalability reasons. This limited number is called information window. When Mosix gathers information about local resources, it selects two machines, one from its window and the other from non-window machines. This mechanism makes Mosix scalable. The main challenge with this mechanism is what size of window is suitable. Simulations show that if there is N machines in the cluster, a window size equal to logN is suitable [3], [5].

One of the major drawbacks of Mosix information dissemination is its periodic approach. The periodic information dissemination can result in waste of network bandwidth and CPU time. On the other hand, if there are many changes in indices during a period, it will result in unsuitable information. Event-driven information dissemination solves these problems [6].

As mentioned earlier, Mosix employs CPU load, memory usage and IPC information to balance and distribute load among machines belonging to a cluster [3]. In the following, we first describe the Mosix basic load balancing mechanism and then discuss its memory sharing and IPC optimization. In general, a load-balancing algorithm must provide four steps: (1) indices computation (2) information dissemination

(3) load balancing trigger, and (4) load balancing operation. In step four, load balancer must decide on migration destination and a candidate process.

CPU load balancing in Mosix operates based on the amount of available CPU for each process. Mosix computes the amount of CPU time taken by each process. It counts the number of executable processes in a period (one second) and normalizes it with CPU speed in favor of maximum CPU speed in the cluster. Then it changes the index in favor of depreciation to realize actual load based on load history. Gathered information is disseminated between some machines. The Mosix load balancing mechanism is triggered when the difference between a local machine's load and the windows machines' loads exceed a threshold. Load balancing pair contains a machine with lower load index that has enough memory space to host a migrant process. Selecting a migrant process is an important stage of load balancing operation. CPU-intensive processes have priority for migration. Mosix limits the number of times a process is allowed to migrate in order to prevent starvation. In this paper we have designed three types of processes, CPU-intensive, memory-intensive and IO-intensive [1], [3], [5], [7].

Thrashing is a phenomenon in operating systems that occurs by the growth of page faults. When free memory in the system is decreased, processes requiring memory to bring in their pages into memory, page fault. Mosix migrates a process if free memory size drops below a threshold. Indeed, this algorithm distributes processes in the memory of the entire cluster. Although, this algorithm results in unbalanced CPU load, but it can increase performance when system into thrashes [3], [4], [8].

When free memory size drops below a threshold, memory sharing algorithm is triggered. When free memory size drops below this threshold, Mosix expects increases in page faults. Therefore, determination of this threshold is a crucial decision. Target machine at least must satisfy two conditions. First, target machine must have enough space to host a new process. Second, target machine must be in window. Mosix usually selects a target machine with the least CPU load. It selects a process with the least migration cost that has caused memory overload. Migration of the selected process must increase free memory size up to threshold and migration must not overload the target machine . If there is no such process, Mosix selects a larger process that can be placed at the target machine. After this replacement, if there is still memory overload, Mosix repeats the above two steps. We have designed a program for this algorithm [3], [8].

The main goal of balancing load based on IPC is to reduce the communication overhead between processes while keeping the load balanced as much as possible. CPU load balancing tries to utilize all processors equally while keeping communicating processes together results in lower communication cost. Therefore, it would reduce response time in the presence of load balancing if processes have low communication. However, in real science applications, communication between processes is high. Therefore, balancing the load based on IPC information can lower the response time. Mosix employs a producer consumer model to optimize its load-balancing algorithm. In this model, each consumer tries to find a producer that presents its product with lower cost. In the cluster environment, products are resources such as CPU cycles, memory and communication channels. Consumers are processes residing on machines. The cost of a resource is the amount of time that a process spends to use one unit of that resource. Whereas a process can run on another machine with lower cost, it is migrated to that machine [3], [9].

Mosix computes the cost of each process based on CPU load, free memory space, and IPC information. It uses a heuristic to compute an approximate response time. The algorithm is initiated when the cost difference between running the process on the current machine and running it on one of the window machines exceeds the migration cost. Mosix selects a process with maximum difference and a machine with the least cost as the target machine [3].

Mosix measures performance in terms of speed of CPU in GHz unit. It also measures the load of each node by counting the number of processes in each scheduling time and computes the primary load as the average of counted processes. It then normalizes the load relative to CPU speedup. A load unit represents 100% utilization of a 'standard' processor [2], [3]. We use these measurements in reporting the results of our experiments in Section III.

III. Evaluation

We have deployed a 5-node cluster of openMosix to analyze the behavior of Mosix in balancing the load on the nodes of the cluster. Each node of cluster had direct access to other nodes through an 8-port 10/100 Ethernet switch. All nodes ran openSUSE 11.2 as their operating system and were equipped with LAM/MPI. Table 1 shows more details of the testbed.

TABLE 1 SPECIFICATION OF THE DEPLOYED OPENMOSIX CLUSTER

System ID	CPU	Memory	OS	MPI
Node1	Intel1.7GHz	256MB	openSUSE11.2	LamMPI
Node2	IntelCeleron 2.4GHz	256MB	openSUSE11.2	LamMPI
Node3	Intel1.8GHz	256MB	openSUSE11.2	LamMPI
Node4	IntelCeleron 2.4GHz	256MB	openSUSE11.2	LamMPI
Node5	IntelCeleron 2.4GHz	1GB	openSUSE11.2	LamMPI

We have designed and ran 7 test programs to examine the behavior of openMosix's load balancing mechanism with different types of node overloading including CPU, I/O, and IPC. We also tested its behavior on MPI directives.

A. *Test No. 1, CPU-intensive*

In this test, we check how openMosix reacts when cluster is overloaded with CPU-intensive processes. We use a process (Figure 1) with an infinite loop that consumes as much CPU cycles as it could.

```
while (1){
        ;
    }
```

Figure 1 - Pseudo code of a CPU-intensive process

We ran 20 instances of this process with *mosrun* command. Table 2 shows the results of these runs; the number in each cell shows the number of instances of the process on that node; we have ignored non-Mosix processes; the *mosrun* command started from Node1. As it is shown in Table 2, openMosix spreads the load on the entire cluster, with respect to nodes' abilities.

TABLE 2 CPU-INTENSIVE TEST RESULTS

	Node1	**Node2**	**Node3**	**Node4**	**Node5**
Startup	20	-	-	-	-
Balanced	4	4	4	4	4

The performance of each node depended on its processor power. Figure 2 shows the performance of cluster on each node.

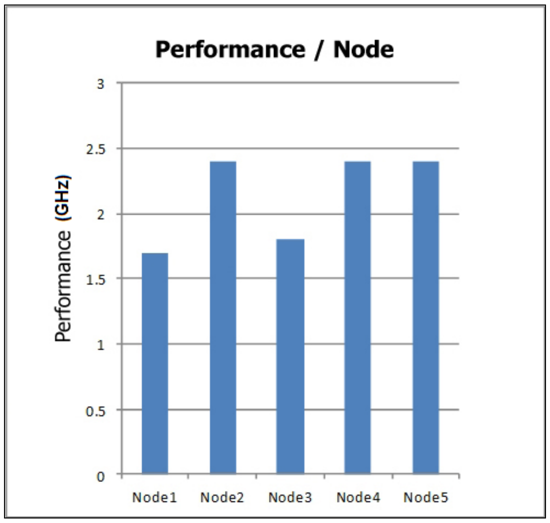

Figure 2 - Performance of nodes

Figure 3 shows the loads on the cluster nodes when 5 instances of a specific process ran on the cluster. Mosix calculates load for each node in the cluster with respect to relative node performance and number of processes in run queue. When numbers of processes on each machine are equal the output load diagram is look like Figure 3.

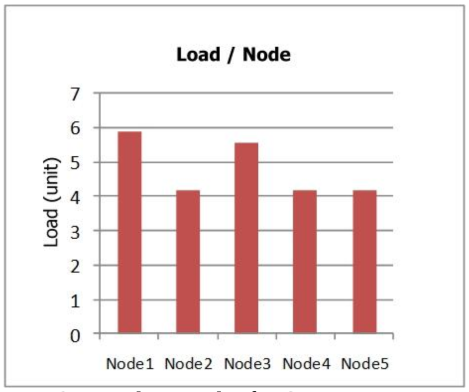

Figure 3 – Loads on nodes for CPU-intensive processes

Normalizing load with respect to node's relative performance, Mosix attempts to overcome the impact of performance heterogeneity. It calculates relative performance through dividing each node's performance by maximum performance in cluster.

B. Test No. 2, I/O-intensive

In this test, we check how openMosix deals with I/O-intensive processes. We use a process (Figure 4) with an infinite loop that sends an empty string to the standard output in each iteration.

```
while(1){
            printf("");
            }
```

Figure 4 – Pseudo code of an I/O-intensive process

After running 20 instances of this I/O-intensive process on the cluster, Mosix migrated them to other nodes with respect to each node's performance. But before migrating each I/O-intensive process, it created a shadow process with a "./" prefix in front of its name on the main host *mosrun* executed. While the process was migrated, the shadow process remained to handle future references of the migrated process to its local host. Due to the existence of these shadow processes and remote references from the migrated process to the local host, Mosix was reluctant to migrate I/O-intensive processes like CPU-intensive processes. Table 3 shows the results of this test, wherein the number in parentheses shows how many shadow processes were located on the local host.

TABLE 3 I/O-INTENSIVE TEST RESULTS

	Node1	**Node2**	**Node3**	**Node4**	**Node5**
Startup	20	-	-	-	-
Balanced	4 (20)	4	4	4	4

C. *Test No. 3, Memory-intensive*

In this test, we checked how openMosix dealt with memory-intensive processes. We used a process (Figure 5) containing CPU-intensive parts because memory-only-intensive processes did not trigger the Mosix load balancer mechanism.

```
for(i=1;i<1000000;i++){
                    malloc(10000);
                           }
while (1){
            ;
             }
```

Figure 5 - Pseudo code of a memory-intensive process

By running the code shown in Figure 5, a large amount of RAM was occupied at the beginning and then a CPU-intensive phase started. In regular situations, Mosix tried to reduce the load on a specific node by migrating some of the processes in that node to other nodes. Nevertheless, in this case it refused to do any migration although the nodes were overloaded. Migration of a large amount of memory is costly. Therefore, Mosix keeps memory-intensive processes in their places as long as the administrator does not force a migration.

D. *Test No. 4, IPC-intensive – Direct*

A process is IPC-intensive when it repeatedly sends messages to other processes. When a process sends too many messages, it becomes a candidate for migration by Mosix. However, as in the case of memory-intensive processes, the migration is costly and Mosix does not migrate them automatically. This reaction is a part of Mosix's policy in dealing with communicating processes.

Figure 6 shows the results of running our test on 17 instances of two IPC-intensive sender and receiver processes.

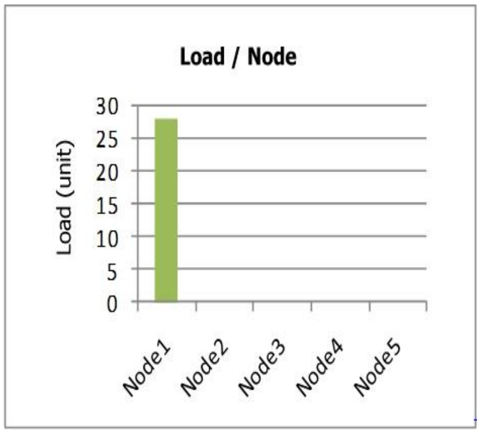

Figure 6 – Loads for IPC-intensive processes before migration

Mosix does not migrate any IPC-intensive processes in this experiment due to their communication cost. Migrating each IPC-intensive process may result in heavy communication cost. So, Mosix attempt to extract process' IPC behaviors and make decision based on it. But when a process passes its IPC age, Mosix does not find its transition soon.

In the next experiment, processes on the first node were migrated to another node manually. After moving all processes to Node 5, the load of the first machine remained almost unchanged (Figure 7), implying that all IPC messages were redirected to the home node.

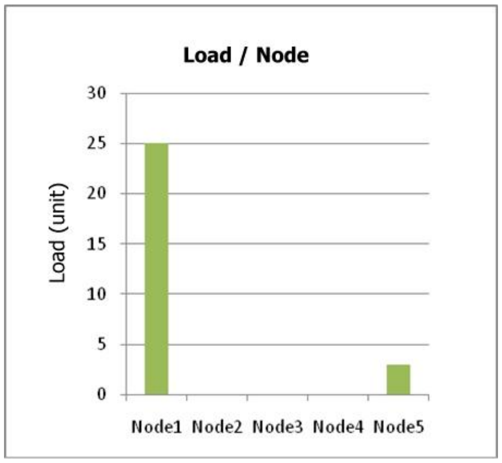

Figure 7 – Loads for IPC-intensive processes after migration

Whereas IPC-intensive processes have heavy communication cost, migrating them does not change their home node's load. Migrated processes communicate with their deputy on home node and communicate via their home node. Therefore, home node's load increases and processes response time falls down.

E. *Test No. 5, IPC-intensive – Shared memory*

Shared memory is another popular IPC mechanism that we had to investigate its support in Mosix. When the "mosrun" command was executed, Mosix returned the error message "*MOSRUN: Attaching SYSV shared-memory not supported under MOSIX (try 'mosrun -e' or 'mosrun -E')*". This error means that MOSIX does not support shared-memory-based communication between processes.

F. *Test No. 6, Forked processes*

A child process inherits the features of its parents. In Mosix, a parent can fork a child that can in turn fork its own child. The hierarchy of parent-child can grow until the number of processes reaches a threshold.

We tested the inheritability of parent features in their children in Mosix and found out that whenever a parent process created a child process, Mosix passed the features of the parent to the child process (Table 4).

TABLE 4 FEATURE INHERITANCE OF FORKED PROCESSES

	Node1	Node2	Node3	Node4	Node5
Startup	1	-	-	-	-
Balanced	*	*	*	*	*

G. Test No. 7, Pipe-based processes

To investigate the behavior of Mosix in pipe-based IPC communications, we tested a pair of producer consumer processes. We initialized the consumer process manually that later created the producer process to provide input for the consumer process.

When the "mosrun" command was executed, Mosix returned the error message "*MOSRUN: Attaching SYSV shared-memory not supported under MOSIX (try 'mosrun -e' or 'mosrun -E')*". This error means that MOSIX does not support pipe-based IPC communication between processes.

We thus used the "mosrun –e" command instead. The results were interesting. After running the test program with "mosrun -e" command, 10 processes were initialized on the first node but after a short time Mosix migrated some of them. Figure 8 shows the cluster status immediately after initializing processes on the first node while Figure 9 shows the cluster status some time after migrations happened and cluster became stable.

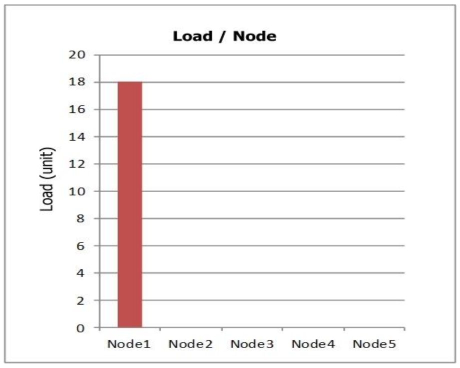

Figure 8 – Loads for Pipe-based processes before migration

Mosix treats piped processes as like as IPC-intensive processes but there is some difference. When processes communicate via pipe, their waiting time is more than the time they uses messages. Since Mosix counts number of processes in ready queue at each scheduling period, it calculates fewer loads.

The interesting point is that although Mosix migrated some processes to other nodes than their home (first node), the load on the home node remained unchanged (Figure 9). This was because copies of migrated processes remained on the home node to communicate with their producer counterparts. We can thus conclude that this migration had been redundant and only had increased the network traffic with undesirable effect on overall performance.

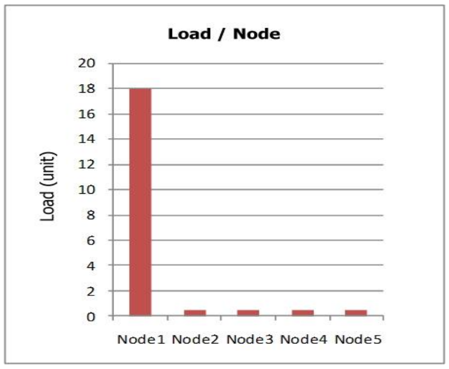

Figure 9 – Loads for Pipe-based processes after migration

By migrating some piped processes to other cluster nodes, communications must take place through communication infrastructures and file system. So, the home node's load does not changed after migrating piped processes.

H. Test No. 8, MPI-based processes

MPI uses sockets and shared memory [10], while Mosix does not efficiently support these two communication mechanisms. Therefore, MPI processes could not be migrated by Mosix. A new "direct communication" feature has been recently added to Mosix that provides migratable sockets for MOSIX processes, but there is still no support for shared memory in Mosix [11], [12].

Therefore, it is impossible to run default MPI-based applications on a Mosix cluster yet. However, there are some short ways. LamMPI, configured with the "--with-rpi=tcp" option can bypass this limitation of Mosix.

In fact, "--with-rpi=tcp" option ensures that no shared memory is used in communications between processes. Therefore, when there is no shared memory in use, Mosix handles an MPI process like any other process.

I. Test No. 9, Priority in migration

To investigate how Mosix prioritizes processes for migration, we ran a number of tests. We tried to identify what processes become candidates for migration by Mosix. We compared two types of processes in each test, but compared all four types of CPU-intensive, I/O-intensive, IPC-intensive, and memory-intensive processes in our final experiment.

In our experiments, Mosix migrated CPU-intensive processes with low allocated memory first. It then migrated I/O-intensive processes and at last equally migrated the IPC-intensive and memory-intensive processes. However, this order was not fixed on all Mosix clusters because of Mosix decision making function. For example, if the power of machines and the amount of available physical memory installed on each machine in a Mosix cluster were widely different, the pattern of Mosix migration priority might be diverse.

IV. Conclusion and Future Works

We ran a number of test programs on a 5-node Mosix cluster to check the real abilities of Mosix in providing a reasonably high performance in handling different requests. We found that Mosix did not guarantee the performance improvement in all cases and that it even reduced the performance by making wrong decisions.

We showed that the Mosix cluster handled CPU-intensive, memory-intensive, and I/O-intensive processes effectively although it was slow and sometimes inaccurate when the cluster was overloaded with large memory-intensive processes. It also properly supported feature inheritance by inheriting all features of parents in the forked children.

We also showed that Mosix did not support shared-memory-based communications between processes and as a result did not support MPI-based processes too unless processes used a different mechanism for their communications than the shared memory. Worst of all, Mosix misbehaved in dealing with pipes and made decided wrongly in migrating tightly-connected processes to other machines, lowering the performance and increasing the network traffic rather than improving the performance.

References

[1] A. Barak, S. Guday, and R. G. Wheeler, The MOSIX Distributed Operating System: Load Balancing for UNIX, Secaucus, Ed. New York, USA: Springer, 1993.

[2] A. Barak and O. La'adan, "The Mosix Multicomputer Operating System for High Performance Cluster Computing," Future Generation Computer Systems, vol. 13, no. 4-5, pp. 361-372, Mar. 1998.

[3] A. Barak, A. Braverman, I. Gilderman, and O. Laadan, "The MOSIX Multicomputer Operating System for Scalable NOW and its Dynamic Resource Sharing Algorithms," The Hebrew University Technical 96-11, 1996.

[4] A. Barak, O. La'adan, and A. Shiloh, "Scalable Cluster Computing with Mosix for Linux," in the 5th Annual Linux Expos, Raleigh, 1999, pp. 95-100.

[5] J. M. Meehan and A. Ritter, "Machine Learning Approach to Tuning Distributed Operating System Load Balancing Algorithms," in Proceedings of the ISCA 19th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS), San Francisco, 2006, pp. 122-127.

[6] M. Beltrán and A. Guzmán, "How to Balance the Load on Heterogeneous Clusters," International Journal of High Performance Computing Applications , vol. 23, no. 1, pp. 99-118, Feb. 2009.

[7] R. Lavi and A. Barak, "The Home Model and Competitive Algorithms for Load Balancing in a Computing Cluster ," in the 21st International Conference on Distributed Computing Systems , Washington DC, Apr, 2001, pp. 127-136.

[8] A. Barak and A. Braverman, "Memory ushering in a scalable computing cluster," Microprocessors and Microsystems , vol. 22, no. 3-4, pp. 175-182, Aug. 1998.

[9] A. Keren and A. Barak, "Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster," IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 1, pp. 39-50, Jan. 2003.

[10] "MPI: A Message-Passing Interface Standard," Message Passing Interface Forum Standard 2.2, 2009.

[11] S. D. S. Pty/Ltd. Mosix updates. http://www.mosix.com.au/updates.html. Jul, 2011, [retrieved: Sep, 2011].

[12] A. S. Amnon Barak. The MOSIX Management System for Linux Clusters, Multi-Clusters, GPU Clusters and Clouds. http://www.mosix.org. Apr, 2010, [retrieved: Sep, 2011].