# MOSIX Evaluation on a Linux Cluster

Najib Kofahi[1], Saeed Al Zahrani[2], and Syed Manzoor Hussain[3]
[1]Department of Computer Sciences, Yarmouk University, Jordan
[2]Saudi Aramco, SA
[3]Dept. of Information and Computer Science, King Fahd University of Petroleum and Minerals, SA

**Abstract:** *Multicomputer Operating System for Unix (MOSIX) is a cluster-computing enhancement of Linux kernel that supports preemptive process migration. It consists of adaptive resource sharing algorithms for high performance scalability by migrating processes across a cluster. Message passing Interface (MPI) is a library standard for writing message passing programs, which has the advantage of portability and ease of use. This paper highlights the advantages of a process migration model to utilize computing resources better and gain considerable speedups in the execution of parallel and multi-tasking applications. We executed several CPU bound tests under MPI and MOSIX. The results of these tests show the advantage of using MOSIX over MPI. At the end of this paper, we present the performance of the executions of those tests, which showed that in some cases improvement in the performance of MOSIX over MPI can reach tens of percents.*

**Keywords:** *High performance computing, performance evaluation, Linux cluster, MOSIX, MPI, process migration.*

*Received July 282004; accepted September 302004*

## 1. Introduction

In recent years, interest in high performance computing has increased [4, 7, 17]. Also, many computer systems supporting high performance computing have emerged. These systems are classified according to their processing power, and their processors interconnection with memory subsystems. In a Network Of Workstations (NOWs) system, where many users need to share system resources, the performance of executing multiple processes can significantly be improved by process migration. Such system can benefit from process migration through initial distribution of processes, to redistribute processes when the system becomes unbalanced or even to relieve a workstation when its owner wishes so. With the increased interest in Linux Clusters [2, 3, 12, 15, 20], which can be in the form of a NOWs, as a cheap solution for high performance and general purpose computing, it becomes a necessity to examine how we can improve the overall utilization of such systems and allow flexible use of idle nodes/workstations. Two models can be used to achieve these goals, process migration and message passing [5, 6, 7, 8, 16, 18, 19].

Process migration model is a model in which a live process is transferred from one system to another. For efficiency reasons, most of the process migration implementations are done in the operating system kernel rather than user space. MOSIX is an example of this model, which will be discussed later in this paper [15].

Message passing model; on the other hand, is mostly implemented in the user space as a group of libraries that application developers link with. This model requires application developers to write their code according to set of standards. Message passing Interface (MPI) and Parallel Virtual Machine (PVM) are two examples of this model. Throughout this paper, we will only discuss MPI as an example of the Message passing model.

MOSIX developers claim in [15] that "in a Linux cluster running MOSIX kernel, there is no need to modify or link applications with any library, or even assign processes to different nodes, just *fork and forget*, MOSIX does it automatically like an SMP". MOSIX can allow any size Linux cluster of x86 workstations and servers to work as single system, which should simplify job submission and system administration [15]. MPI in contrast requires the involvement of application developers to write their code in a way that can utilize all cluster resources. Furthermore, the user is responsible of assigning processes to different nodes. Until now, 9 versions of MOSIX have been developed. We used the latest version of MOSIX that is compatible with LINUX 2.4. The previous 8 versions are as follows:

- First version was started in the year 1977 and finished in 1979 with the name "UNIX with satellite processors". It was compatible with Bell lab's Unix 6.
- Second version was developed in the year 1983 with the name "MOS". It was compatible with Bell lab's Unix 7.
- Third version was started in 1987 and developed in the year 1988 with the name "NSMOS", it was compatible with Bell lab's Unix 7 with some BSD 4.1 extensions.

- Fourth version was developed in the year 1988 with the name "MOSIX" and was compatible with AT&T Unix system V release 2.
- Fifth version was developed in the same year (1988) with the name MOSIX and it was compatible with AT&T Unix system V release 2.
- Sixth version appeared in the year 1989. It was compatible with AT&T Unix system V release 2 running on the machine NS32532.
- Seventh version was developed in the year 1993 and was compatible with BSD/OS.
- Eighth version was completed in the year 1999 and was enhanced to be compatible with LINUX 2.2. It runs on any x86 machines [15].

This paper will study both high performance-computing models and will compare their performance using set of experiments. The paper is organized in six sections. The next section gives a general idea of process migration and an overview of MOSIX and its features as an example of this computing model. Section 3 gives a short overview of the message passing Interface model and its main advantages and disadvantages. Section 4 presents the performance of several testes under MOSIX and MPI. Related work is given in section 5. Conclusions and future work are given in section 6.

## 2. Process Migration

Process migration is the act of transferring a live process from one system to another system in the network. Process migration is mainly used for efficient management of resources in a distributed system [16]. Process migration has different applications including load distribution, fault resilience, and resource sharing. In high performance computing, dynamic process migration is a valuable mechanism to balance the load on different processors [10] to maximize the utilization of resources or to get better program execution performance [7, 19].

Process migration is needed particularly in high performance computing because of the following reasons:

1. *Load balancing*: Workload will be balanced by distributing processes across the network.
2. *Computation speedup*: The total execution performance of a program can be reduced if a single process can be divided into sub-processes that run concurrently on different processors.
3. *Data access*: If the data being used in the computation are huge, it can be more efficient to have a process run remotely, rather than to transfer all the data locally.

### 2.1. MOSIX Overview

A more detailed description of MOSIX can be found in [2, 11, 12, 13, 15]. The overview presented here is mainly from these references. MOSIX is a distributed operating system for clusters that can make a cluster of x86 based Linux nodes run almost like an SMP. It has the advantage of ease-of-use and near optimal performance [15]. MOSIX operates preemptively to the applications and allows the execution of sequential and parallel programs regardless of where the processes are running or what other cluster users are doing [11]. Shortly after the creation of a new process, MOSIX attempts to assign it to the best available node at that time. MOSIX then continues to monitor the new process, as well as all the other processes, and will move them among the nodes to maximize the overall performance. This is done without changing the Linux interface, and users can continue to see and control their processes as they do while running the process on their local node. Users can also monitor the process migration and memory usages on the nodes using OpenMosixView, a cluster management GUI for MOSIX. OpenMosixView supports special fields such as on which node a process was started, on which node it is currently running, the percentage of memory it is using and its current working directory.

The MOSIX technology consists of two main parts [2]: A set of algorithms to allow adaptive resource sharing, and a mechanism to implement a Preemptive Process Migration (PPM). The two parts are implemented at the kernel level in such away that the kernel interface remains unmodified. Therefore, they are completely transparent to the application level [2].

In MOSIX, each process has A Unique Home Node (UHN) referring to the node where the process was created [2]. This node is normally the login node for the application user. Processes that migrate to other nodes use local resources whenever possible, but interact with the user's environment through the UHN. The PPM can migrate any process at any time to any Available node. Usually, migration of processes is based on provided information by one of the resource sharing algorithms. Users, however, can migrate their processes manually and therefore override any automatic system decisions. The process can either initiate a manual migration synchronously or by an explicit request from another process of the same user. Manual process migration can be useful to implement a particular policy or to test different scheduling algorithms.

The Users can run parallel applications by initiating multiple processes in one node, and then allow the system to assign these processes to the best available nodes at that time. If during the execution of the processes new resource become available, then the resource sharing algorithms are designed to utilize these new resources by possible reassignment of the

processes among the other nodes. The ability to assign and reassign processes is particularly important for ease-of-use and to provide an efficient multi-user, time sharing execution environment. PPM is the main tool for the resource management algorithms. As long as the requirements for the resources, such as the CPU or main memory are below certain threshold, the user's processes are confined to the UHN. When the requirements for resources exceed some threshold levels, then some processes may be migrated to other nodes, to take advantage of available remote resources. The overall goal is to maximize the performance of the system by efficient utilization of network wide resources. PPM is also used for hiding system failures from users to create a view of highly available system.

Unlike the master-slave organization between nodes, where there is a central control, in MOSIX each node can operate as an autonomous system. Thus individual nodes in MOSIX can make all their control decisions independently. This design allows a dynamic configuration, where nodes may join or leave the network with minimal disruptions [2]. The algorithms of MOSIX are decentralized; that is, each node is both a master for processes that were created locally and a server for processes that migrated from other nodes. These algorithms are geared for maximal performance, overhead-free scalability, and ease-of-use [2, 9, 13, 15]. MOSIX algorithms use preemptive process migration and have the following properties:

- *Efficient kernel communication*: This feature reduces the overhead of the internal kernel communications, e. g., between the process and its home site, when it is executing in a remote site. This new protocol is specifically useful for a locally distributed system.
- *Network transparency*: For network related operations, the interactive user and the application level programs see a virtual machine that looks like a single machine. The users use the cluster as a single machine, and are not aware of the fact that some of their processes were migrated to other nodes.
- *PPM*: The essential requirement for a process migration is transparency, that is, the functional aspects of the system's behavior should not be altered as a result of the migration of processes. Achieving this transparency requires that the system is able to locate the process and that the process is unaware of the fact that it has been moved from one node to another. In MOSIX these two requirements are achieved by maintaining in the user's workstation a structure called *deputy* that represents the process and interacts with its environment. After a migration, there are no residual dependencies other than at the home workstation. The process resumes its execution in the new site by few page faults, which bring the necessary parts of the program to that site.
- *Dynamic load balancing*: The main concept that involves initiating process migrations is balancing the load across nodes in the cluster. The algorithms respond to variations in the loads of the nodes, the runtime characteristics of the processes, the number of workstations, and their speeds. This feature is useful for sharing resources and distributing work evenly across nodes to maximize overall system utilization.
- *Decentralized control*: Each workstation makes all its own control decisions independently and there are no master-slave relationships between the workstations.

## 2.2. Process Migration in MOSIX

MOSIX supports preemptive process migration. The migrated process continues to interact with its environment regardless of its location. The migrating process is divided into two perspectives, *the user perspective* and *the system perspective*. The user perspective can be migrated and the system perspective, that is UHN dependent, is not migrated as shown in Figure 1.

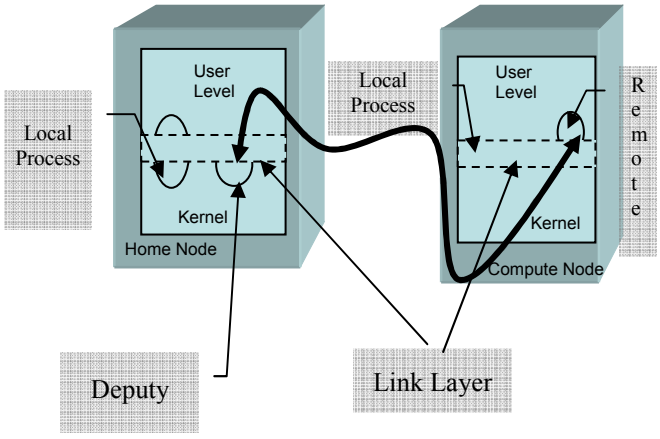


Figure 1. Process migration in MOSIX.

The user perspective, called the *remote*, contains the program code, data, memory maps, stack and registers of the process. The remote encapsulates the process when it is running in the user level. The system perspective, called the *deputy*, contains descriptions of the resources of the process and a kernel-stack for the execution of the system code on behalf of the process. The deputy encapsulates the process when it is running in the kernel. The interface between the remote and the deputy is well defined. Therefore, it is possible to intercept every interaction between the remote and deputy and forward their interactions across the network. This is implemented at the link layer. The deputy holds the site dependent part of the process; hence it must remain in the UHN of that process.

Though the process can migrate many times between the nodes, the deputy is never migrated. Figure 1 shows two processes that share a UHN. The left process is a regular LINUX process while the right process is divided into two halves; *deputy* part remains there while the *remote* part is migrated to another node.

In the execution of a process in MOSIX, location transparency is achieved by forwarding site dependent calls to the deputy at the UHN [2]. The remote site's link layer intercepts all system calls that are executed by the process. These system calls are a synchronous form of interaction between the remote and the deputy. If the system call is site dependent, it will be executed by the remote node locally. Otherwise the system call is forwarded to the deputy, which executes the system call on behalf of the process in the UHN [2].

## 3. Message Passing Interface

MPI is a standard specification for message passing libraries. It is designed to provide a consistent model against which parallel applications can be written for operation of a cluster of workstations. The MPI is widely adopted communication library for parallel and distributed computing [18]. MPI has extremely flexible mechanism for describing data movement routines. It also has a large set of collective computation operations, which allows the users to provide their own set of operations. MPI also provides operations for creating and managing groups in a scalable way. MPI programs can run on network of machines that have different lengths and formats for various fundamental data types. MPI also allows dynamic resizing of cluster nodes. Nodes are allowed to dynamically join and leave a given cluster. In general, MPI has the following features, which makes it appealing for high performance computing:

- *Standardization*: MPI is the only message-passing library that can be considered as a standard. It is supported on virtually all HPC platforms.
- *Portability*: There is no need to greatly modify the source code when porting an application to a different platform, which supports MPI.
- *Performance*: Vendor implementations should be able to exploit native hardware features to optimize performance.
- *Availability*: Variety of implementations are available, both vendor and public domain.

In spite of the above features, Message passing codes are typically very difficult and time consuming to write efficiently. In addition, MPI lacks processor control for multiple programs and it does not specify a standard way of describing the virtual machine upon which an application operates.

## 4. Evaluation

In this section we present a preliminary evaluation of MOSIX and LAM MPI performance of the execution of sets of identical CPU-bound processes. The goal is to highlight the advantages of the MOSIX preemptive process migration mechanism and its load balancing scheme. Several tests were executed, ranging from pure CPU-bound processes in an idle system, to a system with background load. Threads migration performance evaluation was part of what we planned to experiment with but unfortunately threads cannot be migrated in the current release of MOSIX. The granularity of distribution in MOSIX is to the level of process. The distribution of threads is not guaranteed instead. Applications using shared memory also cannot be migrated under the current release of MOSIX.

### 4.1. Experimental Setup

The experimental platform consists of three nodes based on dual Intel Pentium III processors, total of six processors, with 2 GB physical memory and interconnected with a Fast Ethernet network. In the evaluation experiments, we used openmosix4smp kernel level 2.4.18 and LAM MPI version 6.5.6-8.

### 4.2. Performance of CPU-Bound Processes

The first test is intended to show the efficiency of MOSIX load balancing algorithms. We executed a set of identical CPU-bound processes, each requiring 60 seconds, and measured the total execution times under MOSIX (with its preemptive process migration), followed by measurements of the total execution times under MPI.

Table 1 summarizes the results of these tests. In the table, the first column lists the number of processes. The second column lists the measured execution times of the processes using the MOSIX load balancing algorithm. Column three lists the execution times of the same processes under MPI. By comparing column 2 and 3 of Table 1, you can notice that the execution times of MPI (third column) can be significantly slower than MOSIX.

Table 1. MOSIX vs. MPI execution times (sec.).

| No. of Processes | MOSIX (sec.) | MPI (sec.) | MPIS low Down (%) |
|---|---|---|---|
| 1 | 60 | 60 | 0.00 |
| 2 | 59.1 | 60.28 | 2.00 |
| 4 | 60.2 | 65.46 | 8.74 |
| 6 | 59.86 | 69.34 | 15.84 |
| 8 | 61.49 | 75.7 | 23.11 |
| 10 | 60.59 | 85.59 | 41.26 |
| 16 | 60.53 | 90.5 | 49.51 |
| 24 | 64.32 | 140.88 | 119.03 |
| 32 | 69.77 | 168.83 | 141.98 |

Figure 2 depicts the results of Table 1. Comparison of the measured results shows the average slowdown of MPI vs. MOSIX is over 70%, when executing more than 6 processes. This slowdown can become very significant, e. g., 141.98% for 32 processes.

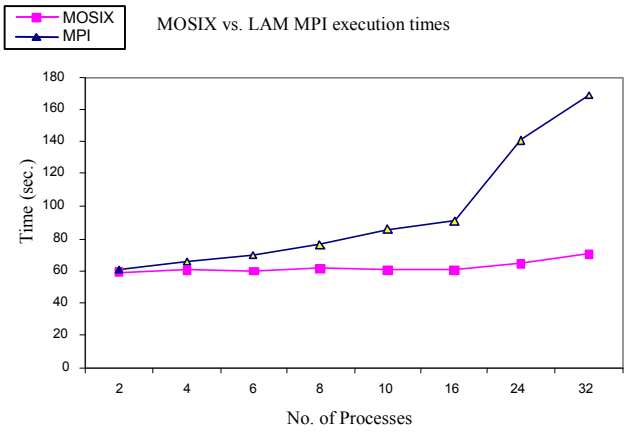


Figure 2. MOSIX vs. MPI execution times.

## 4.3. Performance of CPU-Bound Processes with Background Load

The second test compares the execution times of a set of identical CPU-bound processes under MOSIX and MPI in a system with a background load. This additional load reflects processes of other users in a typical time-sharing computing environment. The specific background load consisted of 4 additional CPU-bound processes that were executed in cycles, where each cycle included an execution period followed by an idle period. The background processes were executed independently, throughout the execution time of the test, and the duration of the execution and idle periods were random variables, in the range of 0 to 15 seconds. In order to get accurate measurements, each test was executed 3 times. Table 2 summarizes the results of these tests and Figure 3 depicts the results of this table.

Table 2. MOSIX vs. MPI with background load execution times.

| No. of Processes | MOSIX (sec.) | MPI (sec.) | MPIS lowness (%) |
|---|---|---|---|
| 2 | 59.61 | 60.08 | 0.79 |
| 4 | 59.93 | 73.04 | 21.88 |
| 6 | 60.38 | 75.62 | 25.24 |
| 8 | 60.55 | 75.19 | 24.18 |
| 10 | 60.57 | 80.42 | 32.77 |
| 16 | 61.21 | 86.06 | 40.60 |
| 24 | 66.96 | 131.65 | 96.61 |
| 32 | 73.62 | 195.15 | 165.08 |

Comparison of the corresponding measured results shows that the average slowdown of MPI vs. MOSIX is over 50%, with as much as 165.08% slowdown, in the measured range, for 32 processes. From these measurements, it follows that in a multi-user environment, when it is expected that background processes of other users are running, execution of parallel programs under MPI may result in a significant slowdown vs. the same execution with MOSIX.

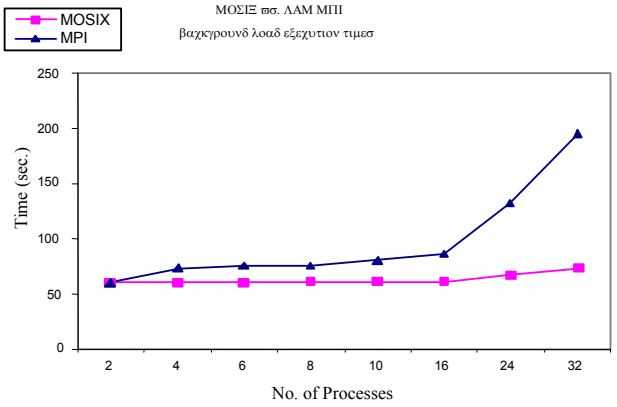


Figure 3. MOSIX vs. MPI with background load execution times.

## 4.4. Performance of Matrix-Multiplication Application

The third test compares the execution times of a real high performance example, matrix multiplication. In this test, we have chosen large matrix sizes, which require each process to do both CPU crunching and relatively high memory utilization for storing the matrices. Figure 4 shows the application performance while running the application using both MOSIX and MPI for a matrix of size 500x500. Figure 5 shows the application performance while running the application using both MOSIX and MPI for a matrix of size 5000x200.

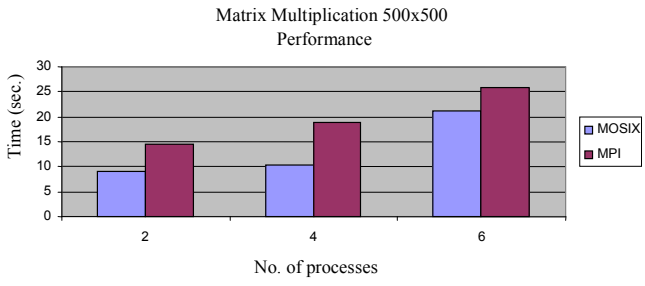


Figure 4. Matrix multiplication 500x500 performance using MOSIX and MPI.

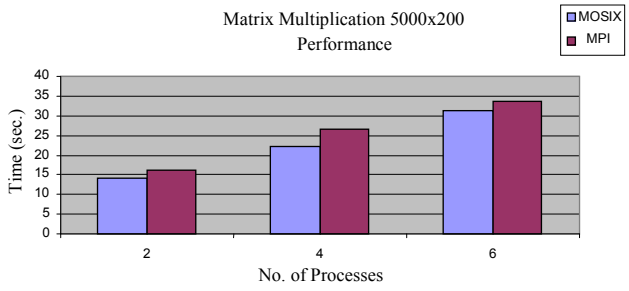


Figure 5. Matrix multiplication 5000x200 performance using MOSIX and MPI.

## 5. Related Work

A comparison of the performance of MOSIX versus other similar systems was carried out by many researchers including [1].

In the early 80's process migration was a new concept and the use of it was considered taboo mainly because of the cost factor, but in the recent past it has developed many folds so that process migration has become an essential part [5, 6, 7, 16, 19]. It is now used hundreds of times daily to provide substantial speed-ups for processes open to parallel processing, such as simulation. However, the usefulness of remote execution would be restricted if processes are to be terminated if they behave differently when they run remotely.

MOSIX provides a transparent process migration facility to allow user processes to migrate without requiring to link with special libraries or usage of system calls. Steve McClure and Richard Wheeler have made related comparisons on various computing nodes to find the best hardware platform for handling Load Balancing through MOSIX [20].

Although many experimental process migration mechanisms have been implemented, MOSIX is one of only a few to receive extensive practical use. Others include SPRITE and LOCUS [5, 6]. SPRITE, developed at UC Berkeley, provides a transparent process migration facility to allow noninvasive access by migrating a remote process during execution if its host becomes unavailable, thus leaving no residual dependencies on the remote host after migration [6].

## 6. Conclusions and Future Work

This paper presented a performance evaluation of MOSIX as an example of a preemptive process migration package compared to LAM MPI. The paper presented the performance of several tests that were developed and executed under MOSIX, and MPI. We showed that in many executions, the performance of applications written in MPI was significantly lower than those executed on MOSIX. For a multi-user environment, MOSIX was proven to be better than MPI in utilizing all system resource because of the load balancing algorithms implemented in MOSIX.

In the future, the performance of MOSIX can be compared with other workload management systems like condor and also one may be interested in running the performance tests on more processors and include communication bound processes test to further compare MOSIX with MPI. Also one can measure the system load/throughput with and without MOSIX.

## References

[1] Aringhieri R., "Open Source Solutions for Optimization on Linux Clusters," *DISMI Technical Report 23*, University of Modena and Reggio Emilia, 2002.

[2] Barak A., Oren L, and Shiloh A., "Scalable Cluster Computing with MOSIX for LINUX," *in Proceedings of the 5th Annual Linux Expo*, Raleigh, N.C., pp. 95-100, May 1999.

[3] Bohringer S., "Building a Diskless Linux Cluster for High Performance Computations from a Standard Linux Distribution," available at: http://www.uni-essen.de/~bt0756/publications/2003-cluster.pdf, April 2003.

[4] Boklund A., Christian Jiresjo, and Stefan Mankefors, "The Story Behind Midnight: A Part Time High Performance Cluster," *in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, Las Vegas, USA, vol. 1, pp.173-178, June 2003.

[5] Douglis F. and John Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 757-785, August 1991.

[6] Douglis F., "Experience with Process Migration in Sprite," *in Proceedings of Workshop on Experience with Building Distributed and Multiprocessor Systems*, Fort Lauderdale, FL, pp. 59-72, October 1989.

[7] Elleuch A. and Muntean T., "Process Migration Protocols for Massively Parallel Systems," *in Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, IEEE Computer Society Press, pp. 84-95, May 1994.

[8] Georg S., "CoCheck: Checkpointing and Process Migration for MPI," *in Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, HI, pp. 526-531, April 1996.

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

[10] Kofahi N. and Rahman Q., "Empirical Study of Variable Granularity and Global Centralized Load Balancing Algorithms," *in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*, Las Vegas, Nevada, USA, CSREA Press, vol. 1, pp. 283-288, 2004.

[11] LINUX Journal, *http://www.linuxjournal.com*.

[12] Lior A., Barak A., and Shiloh A., "The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems," *Cluster Computing*, vol. 7, no. 2, pp. 141-150, April 2004.

[13] Lior A., Barak A., and Shiloh A., "The MOSIX Parallel I/O System for Scalable I/O Performance," *in Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'2002)*, Cambridge, MA, pp. 495-500, November 2002.

[14] MadhuSudhan R. and Kota S., "Infrastructure for Load Balancing on Mosix Cluster," available at: http://www.cis.ksu.edu/~sada/690_897_report.pdf, 2004.

[15] MOSIX, *http://www.mosix.org*, 2003.

[16] Paindaveine Y. and Milojicic D. S., "Process vs. Task Migration," *in Proceedings 29th Hawaii International Conference on System Sciences (HICSS'96)*, Software Technology and Architecture, Maui Hawaii, vol. 1, pp. 636-645, 1996.

[17] Rajkumar B., "A Study on HPC Systems Supporting Single System Image, Techniques and Applications," *in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, CSREA Publishers, Las Vegas, USA, 1997.

[18] Ricky K. K. M., Wang C. L., and Francis C. M. L., "M-Java MPI: A Java-MPI Binding with Process Migration Support," *in Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, Berlin, Germany, pp. 255-263, May 2002.

[19] Roush E. T. and Campbell R. H., "Fast Dynamic Process Migration," *in Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'1996)*, IEEE, pp. 637-645, 1996.

[20] Steve M. and Wheeler R., "MOSIX: How Linux Clusters Solve Real World Problems," *in Proceedings of USENIX Annual Technical Conference*, San Diego, California, USA, June 2000.

**Najib Kofahi** has been an associate professor at Yarmouk University, Jordan, since 1987. He received his PhD from University of Missouri-Rolla, USA, in 1987. He worked as a visiting associate professor at King Fahd University of Petroleum and Minerals (FUPM) during the academic years 2000 - 2003. While at KFUPM, he led an online course development team to develop online course material for algorithms course. He worked extensively in computer science curriculum development at Yarmouk University since his appointment and at Philadelphia University, Jordan in the academic year 1993-1994. At Philadelphia University, he worked as the chairman of Computer Science Department, while being on his first sabbatical leave from Yarmouk University. He worked as chairman of the Computer Science Department at Yarmouk University in the years 1990-1992. He has several journal and conference research publications. His research interests include operating systems, computer system security, and computer applications.

**Saeed Al Zahrani** is a high performance computing system administrator working in Dhahran, Saudi Arabia. He holds a bachelor degree in computer engineering from Oregon State University. Currently he is working for Saudi Aramco. He has more than 6 years of experience in the area of high performance computing with specialty in linux clusters.

**Syed Manzoor Hussain** is a MSc student and a research assistant in the Information and Computer Science Department at King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia. He obtained his BE degree in computer science from Gulbarga University, India in 2002 and joined KFUPM in February 2003. His research interests include software metrics, architectural stability, and operating systems. His current research projects include package cohesion metrics.