Scalable Cluster Computing with MOSIX for LINUX

Amnon Barak *Oren La'adan* *Amnon Shiloh*

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
http://www.mosix.cs.huji.ac.il

ABSTRACT

Mosix is a software tool for supporting cluster computing. It consists of kernel-level, adaptive resource sharing algorithms that are geared for high performance, overhead-free scalability and ease-of-use of a scalable computing cluster. The core of the Mosix technology is the capability of multiple workstations and servers (nodes) to work cooperatively as if part of a single system.

The algorithms of Mosix are designed to respond to variations in the resource usage among the nodes by migrating processes from one node to another, preemptively and transparently, for load-balancing and to prevent memory depletion at any node. Mosix is scalable and it attempts to improve the overall performance by dynamic distribution and redistribution of the workload and the resources among the nodes of a computing-cluster of any size. Mosix conveniently supports a multi-user time-sharing environment for the execution of both sequential and parallel tasks.

So far Mosix was developed 7 times, for different version of Unix, BSD and most recently for Linux. This paper describes the 7-th version of Mosix, for Linux.

1 Introduction

This paper describes the Mosix technology for Cluster Computing (CC). Mosix [4, 5] is a set of adaptive resource sharing algorithms that are geared for performance scalability in a CC of any size, where the only shared component is the network. The core of the Mosix technology is the capability of multiple nodes (workstations and servers, including SMP's) to work cooperatively as if part of a single system.

In order to understand what Mosix does, let us compare a Shared Memory (SMP) multicomputer and a CC. In an SMP system, several processors share the memory. The main advantages are increased processing volume and fast communication between the processes (via the shared memory). SMP's can handle many simultaneously running processes, with efficient resource allocation and sharing. Any time a process is started, finished, or changes its computational profile, the system adapt instantaneously to the resulting execution environment. The user is not involved and in most cases does not even know about such activities.

Unlike SMP's, Computing Clusters (CC) are made of collections of share-nothing workstations and (even SMP) servers (nodes), with different speeds and memory sizes, possibly from different generations. Most often, CC's are geared for multi-user, time-sharing environments. In CC systems the user is responsible to allocate the processes to the nodes and to manage the cluster resources. In many CC systems, even though all the nodes run the same operating system, cooperation between the nodes is rather limited because most of the operating system's services are locally confined to each node. The main software packages for process allocation in CC's are PVM [8] and MPI [9]. LSF [7] and Extreme Linux [10] provide similar services. These packages provide an execution environment that requires an adaptation of the application and the user's awareness. They include tools for initial (fixed) assignment of processes to nodes, which sometimes use load considerations, while ignoring the availability of other resources, e.g. free memory and I/O overheads. These packages run at the user level, just like ordinary applications, thus are incapable to respond to fluctuations of the load or other resources, or to redistribute the workload adaptively.

In practice, the resource allocation problem is much more complex because there are many (different) kinds of resources, e.g., CPU, memory, I/O, Inter Process Communication (IPC), etc, where each resource is used in a different manner and in most cases its usage is unpredictable. Further complexity results from the fact that different users do not coordinate their activities. Thus even if one knows how to optimize the allocation of resources to processes, the activities of other users are most likely to interfere with this optimization.

For the user, SMP systems guarantee efficient, balanced use of the resources among all the running processes, regardless of the resource requirements. SMP's are easy to use because they employ adaptive resource management, that is completely transparent to the user. Current CC's lack such capabilities. They rely on user's controlled static allocation, which is inconvenient and may lead to significant performance penalties due to load imbalances.

Mosix is a set of algorithms that support adaptive resource sharing in a scalable CC by dynamic process migration. It can be viewed as a tool that takes CC platforms one step closer towards SMP environments. By being able to allocate resources globally, and distribute the workload dynamically and efficiently, it simplifies the use of CC's by relieving the user from the burden of managing the cluster-wide resources. This is particularly evident in a multi-user, time-sharing environments and in non-uniform CC's.

2 What is Mosix

Mosix [4, 5] is a tool for a Unix-like kernel, such as Linux, consisting of adaptive resource sharing algorithms. It allows multiple Uni-processors (UP) and SMP's (nodes) running the same kernel to work in close cooperation. The resource sharing algorithms of Mosix are designed to respond on-line to variations in the resource usage among the nodes. This is achieved by migrating processes from one node to another, preemptively and transparently, for load-balancing and to prevent thrashing due to memory swapping. The goal is to improve the overall (cluster-wide) performance and to create a convenient multi-user, time-sharing environment for the execution of both sequential and parallel applications. The standard run time environment of Mosix is a CC, in which the cluster-wide resources are available to each node. By disabling the automatic process migration, the user can switch the configuration into a plain CC, or even an MPP (single-user) mode.

The current implementation of Mosix is designed to run on clusters of X86/Pentium based workstations, both UP's and SMP's that are connected by standard LANs. Possible configurations may range from a small cluster of PC's that are connected by Ethernet, to a high performance system, with a large number of high-end, Pentium based SMP servers that are connected by a Gigabit LAN, e.g. Myrinet [6].

2.1 The technology

The Mosix technology consists of two parts: a Preemptive Process Migration (PPM) mechanism and a set of algorithms for adaptive resource sharing. Both parts are implemented at the kernel level, using a loadable module, such that the kernel interface remains unmodified. Thus they are completely transparent to the application level.

The PPM can migrate any process, at any time, to any available node. Usually, migrations are based on information provided by one of the resource sharing algorithms, but users may override any automatic system-decisions and migrate their processes manually. Such a manual migration can either be initiated by the process synchronously or by an explicit request from another process of the same user (or the super-user). Manual process migration can be useful to implement a particular policy or to test different scheduling algorithms. We note that the super-user has additional privileges regarding the PPM, such as defining general policies, as well as which nodes are available for migration.

Each process has a Unique Home-Node (UHN) where it was created. Normally this is the node to which the user has logged-in. In PVM this is the node where the task was spawned by the PVM daemon. The system image model of Mosix is a CC, in which every process seems to run at its UHN, and all the processes of a users' session share the execution environment of the UHN. Processes that migrate to other (remote) nodes use local (in the remote node) resources whenever possible, but interact with the user's environment through the UHN. For example, assume that a user launches several processes, some of which migrate away from the UHN. If the user executes “ps”, it will report the status of all the processes, including processes that are executing on remote nodes. If one of the migrated processes reads the current time, i.e. invokes *gettimeofday()*, it will get the current time at the UHN.

The PPM is the main tool for the resource management algorithms. As long as the requirements for 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 by efficient utilization of the network-wide resources.

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

Mosix has no central control or master-slave relationship between nodes: each node can operate as an autonomous system, and it makes all its control decisions independently. This design allows a dynamic configuration, where nodes may join or leave the network with minimal disruptions. Algorithms for scalability ensure that the system runs well on large configurations as it does on small configurations. Scalability is achieved by incorporating randomness in the system control algorithms, where each node bases its decisions on partial knowledge about the state of the other nodes, and does not even attempt to determine the overall state of the cluster or any particular node. For example, in the probabilistic information dissemination algorithm [4], each node sends, at regular intervals, information about its available resources to a randomly chosen subset of other nodes. At the same time it maintains a small "window", with the most recently arrived information. This scheme supports scaling, even information dissemination and dynamic configurations.

2.2 The resource sharing algorithms

The main resource sharing algorithms of Mosix are the load-balancing and the memory ushering. The dynamic load-balancing algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from higher loaded to less loaded nodes. This scheme is decentralized – all the nodes execute the same algorithms, and the reduction of the load differences is performed independently by pairs of nodes. The number of processors at each node and their speed are important factors for the load-balancing algorithm. This algorithm responds to changes in the loads of the nodes or the runtime characteristics of the processes. It prevails as long as there is no extreme shortage of other resources, e.g., free memory or empty process slots.

The memory ushering (depletion prevention) algorithm is geared to place the maximal number of processes in the cluster-wide RAM, to avoid as much as possible thrashing or the swapping out of processes [2]. The algorithm is triggered when a node starts excessive paging due to shortage of free memory. In this case the algorithm overrides the load-balancing algorithm and attempts to migrate a process to a node which has sufficient free memory, even if this migration would result in an uneven load distribution.

3 Process migration

Mosix supports preemptive (completely transparent) process migration (PPM). After a migration, a process continues to interact with its environment regardless of its location. To implement the PPM, the migrating process is divided into two contexts: the user context – that can be migrated, and the system context – that is UHN dependent, and may not be migrated.

The user context, called the *remote*, contains the program code, stack, data, memory-maps and registers of the process. The *remote* encapsulates the process when it is running in the user level. The system context, called the *deputy*, contains description of the resources which the process is attached to, and a kernel-stack for the execution of system code on behalf of the process. The *deputy* encapsulates the process when it is running in the kernel. It holds the site-dependent part of the system context of the process, hence it must remain in the UHN of the process. While the process can migrate many times between different nodes, the *deputy* is never migrated.

The interface between the user-context and the system context is well defined. Therefore it is possible to intercept every interaction between these contexts, and forward this interaction across the network. This is implemented at the link layer, with a special communication channel for interaction. Figure 1 shows two

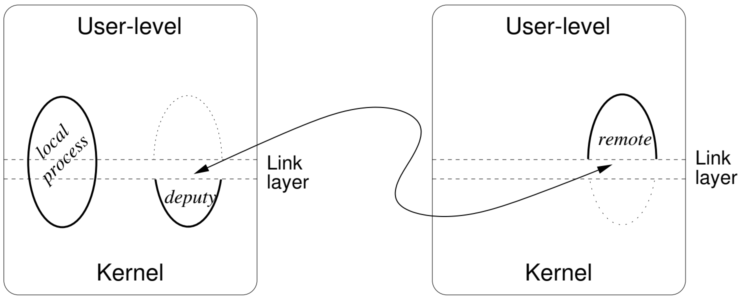

Figure 1: A local process and a migrated process

processes that share a UHN. In the figure, the left process is a regular Linux process while the right process is split, with its *remote* part migrated to another node.

The migration time has a fixed component, for establishing a new process frame in the new (remote) site, and a linear component, proportional to the number of memory pages to be transfered. To minimize the migration overhead, only the page tables and the process' dirty pages are transferred.

In the execution of a process in Mosix, location transparency is achieved by forwarding site dependent system calls to the *deputy* at the UHN. System calls are a synchronous form of interaction between the two process contexts. All system calls that are executed by the process are intercepted by the remote site's link layer. If the system call is site independent it is executed by the *remote* locally (at the remote site). Otherwise, the system call is forwarded to the *deputy*, which executes the system call on behalf of the process in the UHN. The *deputy* returns the result(s) back to the remote site, which then continues to execute the user's code.

Other forms of interaction between the two process contexts are signal delivery and process wakeup events, e.g. when network data arrives. These events require that the *deputy* asynchronously locate and interact with the *remote*. This location requirement is met by the communication channel between them. In a typical scenario, the kernel at the UHN informs the *deputy* of the event. The *deputy* checks whether any action needs to be taken, and if so, informs the *remote*. The *remote* monitors the communication channel for reports of asynchronous events, e.g., signals, just before resuming user-level execution. We note that this approach is robust, and is not affected even by major modifications of the kernel. It relies on almost no machine dependent features of the kernel, and thus does not hinder porting to different architectures.

One drawback of the *deputy* approach is the extra overhead in the execution of system calls. Additional overhead is incurred on file and network access operations. For example, all network links (sockets) are created in the UHN, thus imposing communication overhead if the processes migrate away from the UHN. To overcome this problem we are developing “migratable sockets”, which will move with the process, and thus allow a direct link between migrated processes. Currently, this overhead can significantly be reduced by initial distribution of communicating processes to different nodes, e.g. using PVM/MPI. Should the system become imbalanced, the Mosix algorithms will reassign the processes to improve the performance [3].

4 The implementation

The porting of Mosix for Linux started by a feasibility study. We also developed an interactive kernel debugger, a pre-requisite for any project of this scope. The debugger is invoked either by a user request, or when the kernel crashes. It allows the developer to examine kernel memory, processes, stack contents, etc. It also allows to trace system calls and processes from within the kernel, and even insert break-points in the kernel code.

In the main part of the project, we implemented the code to support the transparent operation of split processes, with the user-context running on a *remote* node, supported by the *deputy*, which runs in the UHN. At the same time, we wrote the communication layer that connects between the two process contexts and designed their interaction protocol. The link between the two contexts was implemented on top of a simple, but exclusive TCP/IP connection. After that, we implemented the process migration mechanism, including migration away from the UHN, back to the UHN and between two remote sites. Then, the information dissemination module was ported enabling exchange of status information among the nodes. Using this facility, the algorithms for process-assessment and automatic migration were also ported. Finally, we designed and implemented the Mosix application programming interface (API) via the `/proc`.

4.1 Deputy / Remote mechanisms

The *deputy* is the representative of the *remote* process at the UHN. Since the entire user space memory resides at the remote node, the *deputy* does not hold a memory map of its own. Instead, it shares the main kernel map similarly to a kernel thread.

In many kernel activities, such as the execution of system calls, it is necessary to transfer data between the user space and the kernel. This is normally done by the `copy_to_user()`, `copy_from_user()` kernel primitives. In Mosix, any kernel memory operation that involves access to user space, requires the *deputy* to communicate with its *remote* to transfer the necessary data.

The overhead of the communication due to remote copy operations, which may be repeated several times within a single system call, could be quite substantial, mainly due to the network latency. In order to eliminate excessive remote copies, which are very common, we implemented a special cache that reduces the number of required interactions by prefetching as much data as possible during the initial system call request, while buffering partial data at the *deputy* to be returned to the *remote* at the end of the system call.

To prevent the deletion or overriding of memory-mapped files (for demand-paging) in the absence of a memory map, the *deputy* holds a special table of such files that are mapped to the *remote* memory. The user registers of migrated processes are normally under the responsibility of the *remote* context. However, each register or combination of registers, may become temporarily owned for manipulation by the *deputy*.

Remote (guest) processes are not accessible to the other processes that run at the same node (locally or originated from other nodes) - and vice versa. They do not belong to any particular user (on the remote node, where they run) nor can they be sent signals or otherwise manipulated by local processes. Their memory can not be accessed and they can only be forced, by the local system administrator, to migrate out.

A process may need to perform some Mosix functions while logically stopped or sleeping.

Such processes would run Mosix functions "in their sleep", then resume sleeping, unless the event they were waiting for has meanwhile occurred. An example is process migration, possibly done while the process is sleeping. For this purpose, Mosix maintains a logical state, describing how other processes should see the process, as opposed to its immediate state.

4.2 Migration constraints

Certain functions of the Linux kernel are not compatible with process context division. Some obvious examples are direct manipulations of I/O devices, e.g., direct access to privileged bus-I/O instructions, or direct access to device memory. Other examples include writable shared memory and real time scheduling. The last case is not allowed because one can not guarantee it while migrating, as well as being unfair towards processes of other nodes.

A process that uses any of the above is automatically confined to its UHN. If the process has already been migrated, it is first migrated back to the UHN.

4.3 Information collection

Statistics about a process' behavior are collected regularly, such as at every system call and every time the process accesses user data. This information is used to assess whether the process should be migrated from the UHN. These statistics decay in time, to adjust for processes that change their execution profile. They are also cleared completely on the "execve()" system call, since the process is likely to change its nature.

Each process has some control over the collection and decay of its statistics. For instance, a process may complete a stage knowing that its characteristics are about to change, or it may cyclically alternate between a combination of computation and I/O.

4.4 The Mosix API

The Mosix API has been traditionally implemented via a set of reserved system calls, that were used to configure, query and operate Mosix. In line with the Linux convention, we modified the API to be interfaced via the `/proc` file system. This also prevents possible binary incompatibilities of user programs between different Linux versions.

The API was implemented by extending the Linux `/proc` file system tree with a new directory /proc/mosix. The calls to Mosix via `/proc` include: synchronous and asynchronous migration requests; locking a process against automatic migrations; finding where the process currently runs; finding about migration constrains; system setup and administration; controlling statistic collection and decay; information about available resources on all configured nodes, and information about remote processes.

5 Conclusions

Mosix brings the new dimension of scaling to cluster computing with Linux. It allows the construction of a high-performance, scalable CC from commodity components, where scaling does not introduce any performance overhead. The main advantage of Mosix over other CC systems is its ability to respond at run-time to unpredictable and irregular resource requirements by many users.

The most noticeable properties of executing applications on Mosix are its adaptive resource distribution policy, the symmetry and flexibility of its configuration. The combined effect of these properties implies that users do not have to know the current state of the resource usage of the various nodes, or even their number. Parallel applications can be executed by allowing Mosix to assign and reassign the processes to the best possible nodes, almost like an SMP.

The Mosix R&D project is expanding in several directions. We already completed the design of *migratable sockets*, which will reduce the inter process communication overhead. A similar optimization is *migratable temporary files*, which will allow a *remote* process, e.g. a compiler, to create temporary files in the remote node. The general concept of these optimization is to migrate more resources with the process, to reduce remote access overhead.

In another project, we are developing new competitive algorithms for adaptive resource management that can handle different kind of resources, e.g., CPU, memory, IPC and I/O [1]. We are also researching algorithms for network RAM, in which a large process can utilize available memory in several nodes. The idea is to spread the process's data among many nodes, and rather migrate the (usually small) process to the data than bring the data to the process.

In the future, we consider extending Mosix to other platforms, e.g., DEC's Alpha or SUN's Sparc. Details about the current state of Mosix are available at URL `http://www.mosix.cs.huji.ac.il`.

References

[1] Y. Amir, B. Averbuch, A. Barak, R.S. Borgstrom, and A. Keren. An Opportunity Cost Approach for Job Assignment and Reassignment in a Scalable Computing Cluster. In *Proc. PDCS '98*, Oct. 1998.

[2] A. Barak and A. Braverman. Memory Ushering in a Scalable Computing Cluster. *Journal of Microprocessors and Microsystems*, 22(3-4), Aug. 1998.

[3] A. Barak, A. Braverman, I. Gilderman, and O. Laden. Performance of PVM with the MOSIX Preemptive Process Migration. In *Proc. Seventh Israeli Conf. on Computer Systems and Software Engineering*, pages 38–45, June 1996.

[4] A. Barak, S. Guday, and R.G. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX. In *Lecture Notes in Computer Science, Vol. 672*. Springer-Verlag, 1993.

[5] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[6] N.J. Boden, D. Cohen, R.E. Felderman, A.K. Kulawik, C.L. Seitz, J.N.Seizovic, and W-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[7] Platform Computing Corp. *LSF Suite 3.2*. 1998.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM - Parallel Virtual Machine*. MIT Press, Cambridge, MA, 1994.

[9] W. Gropp, E. Lust, and A.Skjellum. *Using MPI*. MIT Press, Cambridge, MA, 1994.

[10] Red Hat. *Extreme Linux*. 1998.