OpenMosix

Presented by Dr. Moshe Bar and MAASK [01]

openMosix is a kernel extension for single-system image clustering. openMosix [24] is a tool for a Unix-like kernel, such as Linux, consisting of adaptive resource sharing algorithms. It allows multiple uniprocessors (UP) and symmetric multiprocessors (SMP nodes) running the same kernel to work in close cooperation. The openMosix resource sharing algorithms 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 cluster-wide performance and to create a convenient multiuser, time-sharing environment for the execution of both sequential and parallel applications. The standard runtime environment of openMosix is a computing cluster, in which the cluster-wide resources are available to each node.

The current implementation of openMosix is designed to run on clusters of X86/Pentium-based workstations, both UPs and SMPs, that are connected by standard LANs. Possible configurations may range from a small cluster of PCs that are connected by 10Mbps Ethernet, to a high performance system, with a large number of high-end, Pentium-based SMP servers that are connected by a Gigabit LAN, ATM, or Myrinet.

1.1.2 openMosix Technology

The openMosix 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 anytime, 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.

Each process has a Unique Home-Node (UHN) where it was created. Normally this is

the node to which the user has logged-in. The single system image model of openMosix is a CC (cache coherent) cluster, in which every process seems to run at its UHN, and all the processes of a user's 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.

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 a 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 openMosix is the process. 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 resources become available, the resource sharing algorithms are designed to utilize these new resources by possible reassignment of the processes among the nodes. This capability to assign and reassign processes is particularly important for ease-of-use and to provide an efficient multi-user, time-sharing execution environment.

openMosix 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. Additionally, this allows for a very great scalability and ensures 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, 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.

1.1.2.1 The Resource Sharing Algorithms

The main resource sharing algorithms of openMosix 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 such as free memory or empty process slots.

There are two main resource-sharing algorithms in openMosix:

The first one, 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. 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 that has sufficient free memory, even if this migration would result in an uneven load distribution.

Lately, openMosix, was given a new algorithm to select on which node a given program should run. The mathematical model for this scheduling algorithm comes from the field of economics research. Determining the optimal location for a job is a complicated problem. The most important complication is that the resources available on a cluster of Linux computers are heterogeneous. In effect, the costs for memory, CPU, process communication, and so forth are incomparable. They are not even measured in the same units. Communication resources are measured in terms of bandwidth, memory in terms of space, and CPU in terms of cycles. The natural greedy

strategy, balancing the resources across all of the machines, is not even well defined. The new algorithm employed by openMosix is very interesting because it tries to reconcile these differences (and maybe it could be applied to non-cluster schedulers as well) based on economic principles and competitive analysis.

The key idea of this strategy is to convert the total usage of several heterogeneous resources, such as memory and CPU, into a single homogeneous "cost". Jobs are then assigned to the machine where they have the lowest cost. Just like in a market-oriented economy.

This economic strategy provides a unified algorithm framework for allocation of computation, communication, memory, and I/O resources. It allows the development of near-optimal on-line algorithms for allocating and sharing these resources.

1.1.2.2 Process Migration

openMosix supports preemptive and completely transparent process migration (PPM). After 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, which can be migrated, and the system context, which 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 a description of the resources that 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.

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 transferred. To minimize the migration overhead, only the page tables and the process' dirty pages are transferred.

In the execution of a process in openMosix, 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, such as 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, like signals, just before resuming user-level execution. 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.

1.1.2.3 File Access

openMosix has its own new Cluster File System for Linux that gives a shared cluster-wise view of all the file systems. Cluster developers saw that all current solutions for a cluster-wide file system relied on a central file server, but that some new file system

technologies were being developed addressing the very needs of a single system image cluster (SSI) like openMosix.

openMosix uses Direct File System Access (DFSA). DFSA was designed to reduce the extra overhead of executing I/O oriented system-calls of a migrated process. This was done by allowing the execution of most such system-calls locally - in the process's current node. In addition to DFSA, a new algorithm that takes into account I/O operation was added to the openMosix process distribution (load-balancing) policy. The outcome of these provisions is that a process that performs moderate to high volume of I/O is encouraged to migrate to the node in which it does most of its I/O. one obvious advantage is that I/O-bound processes have greater flexibility to migrate from their respective home-nodes for better load-balancing. So, unlike all existing network file systems (say, NFS) which bring the data from the file server to the client node over the network, the openMosix cluster attempts to migrate the process to the node in which the file actually resides.

1.1.2.4 Deputy/Remote Mechanisms

The deputy is the representative of the remote process at the UHN. Because 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 openMosix, 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, a special cache was implemented that reduces the number of required interactions by pre-fetching 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 cannot be accessed and they can only be forced, by the local system administrator, to migrate out.

A process may need to perform some openMosix functions while logically stopped or sleeping.
Such processes would run openMosix functions "in their sleep," and 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, openMosix maintains a logical state, describing how other processes should see the process, as opposed to its immediate state.

1.1.2.5 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 because 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.

1.1.2.6 The openMosix API

The openMosix API has been traditionally implemented via a set of reserved system calls that were used to configure, query, and operate openMosix. In line with the Linux convention, the API was modified 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/openMosix. The calls to openMosix via /proc include: synchronous and asynchronous migration requests; locking a process against automatic migrations; finding where the process currently runs; finding out about migration constraints; system setup and administration; controlling statistic collection and decay; information about available resources on all configured nodes; and information about remote processes.

1.1.3 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, such as 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 cannot guarantee it while migrating, as well as being unfair towards processes of other nodes.
A process that uses any of these functions is automatically confined to its UHN. If the process has already been migrated, it is first migrated back to the UHN.

1.1.4 The openMosix Performance

Unlike MPPs, which allow a single user per partition, CCs are geared for multiuser, timesharing environments. In order to make CC systems as easy to program, manage, and use as an SMP, it is necessary to develop means for global (cluster-wide) resource allocation and sharing that can respond to resource availability, distribute the workload dynamically, and utilize the available, cluster-wide resources efficiently and transparently. Such mechanisms are necessary for performance scalability in clusters of servers and to support a flexible use of workstations because the overall available

resources in such systems are expected to be much larger than the available resources at any workstation or server. The development of such mechanisms is particularly important to support multiuser, time-sharing parallel execution environments, where it is necessary to share the resources and at the same time distribute the workload dynamically, to utilize the global resources efficiently.

1.2. Shared Memory

In Linux shared memory is implemented in two ways:

a. Sys V Inter Process Communication (IPC) Shared Memory

b. Threads (Light weight processes in Linux)

1.2.1 Sys V IPC Shared Memory

Explicitly acquired System V IPC shared memory [6],[7],[8] is the fastest and the most useful means of IPC. It allows two or more processes to access some common data structures by placing them in a shared memory segment.

Shared memory is implemented as a file under the 'Virtual Memory File system (tmpfs or the former shmfs).

[01]Maya, Anu, Asmita, Snehal, Krushna (MAASK)

"MigShm: Shared memory over openMosix" A project report on MigShm

(April 2003)

Formal references included in the complete document available at:
http://mcaserta.com/maask/Migshm_Report.pdf