The OpenMosix Cluster

General Aspects

A openMosix cluster of Linux machines is what is called a Single System Image cluster (SSI). The first SSI clusters were the IBM SysPlex cluster and the VAXCluster. In a VAXCluster you could telnet to cluster.address and you would end up in one of the nodes of the cluster. You, the user, would not care about which node you run, and any program you launched would be executed on the node which best could serve the program's needs, automatically load-balancing the cluster. openMosix is to Linux what the VAXCluster is to VMS.

OpenMosix, is an extension of the kernel. Therefore, to install OpenMosix, you have to apply a patch to the standard Linux kernel sources. The changes make about 3% of all the source code of the kernel, so it's really not that much.

Once you recompiled your kernel and booted with it, you have a working OpenMosix node of a cluster. Replicating this work among all your Linux machines in your network will create an openMosix cluster. Notice that openMosix is available for all versions of 2.4.

As of the openMosix release 2.4.19 there is no configuration file whatsoever, you just boot your cluster and the nodes automagically discover each other.

The nice thing about OpenMosix is that you just launch a program and the cluster decides (unless told otherwise) where to run it. OpenMosix has its roots in MOSIX where the author of this paper was co-project manager for a number of years together with Prof. Amnon Barak at the Hebrew University of Jerusalem.

OpenMosix brings the new dimension of scaling to cluster computing with Linux. It allows the construction of a high-performance, scalable cluster from commodity components, where scaling does not introduce any performance overhead. The main advantage of OpenMosix over other other types of clusters, like the Beowulf lies in its ability to respond dynamically at runtime to unpredictable and irregular resource requirements by many users.

The most noticeable properties of executing applications on OpenMosix are its adaptive resource distribution policy and 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 OpenMosix to assign and reassign the processes to the best possible nodes, almost like an SMP. This re-assignment is called *process migration* in computer science. As such an openMosix cluster is really a High Throughput Cluster.

There are, however, a few areas where the OpenMosix might not create any noticeable benefit. For applications using shared memory, for example, no process migration can happen because all processes accessing said shared memory must reside on the same node.

The openMosix project is a fully GPL2 compliant software hosted and managed on the Sourceforge website. Contributions will be accepted from anyone and there are a number of committers to the source on the CVS. The openMosix project is expanding in several directions. Next to above-mentioned node-autodiscovery and auto-configuration, we have implemented an abstraction for the protocol, process migration and I/O delivery to facilitate support for new types of inter-connects like Dolphin, InfiniBand, Quadrics and others. Right now we have Dolphin working which thanks to

its ultra-low latency (1.4 microseconds transfer time for 4bytes) has given a big boost to the overall performance and scalability. The team has already completed the pre-design of migrateable sockets, which will reduce the inter-process communication overhead. A similar optimization is migrateable temporary files, which will allow a remote process (such as a compiler) to create temporary files in the remote node. Finally, we are also implementing a first version of distributed shared memory (DSM) which will allow openMosix to scale up modern applications like Apache, databases and application servers.

The general concept of these optimizations is to migrate more resources with the process and to reduce remote access overhead.

Finally, the development team is also porting openMosix to the 64bit IA64 platform (Itanium and McKinley). The new platform will finally allow for much bigger address spaces.

OpenMosix in Detail

OpenMosix' core are the adaptive resource sharing algorithms. They allows multiple uni-processors (UP) and SMPs (nodes) running the same kernel to work in close cooperation. The OpenMosix resource sharing algorithms are designed to respond online 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 current implementation of OpenMosix is designed to run on the IA32 architecture, 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,Dolphin or Quadrics.

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. Therefore all the openMosix functionaliy remains 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. 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. It was noted 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. 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. 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 by invoking *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 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, and perhaps more importantly, 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.

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 Linux.

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.

Process Migration

OpenMosix supports preemptive and 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, 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. 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 the team at Hebrew University is developing "migrateable 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, such as using PVM (discussed elsewhere in this book). Should the system become imbalanced, the OpenMosix algorithms will reassign the processes to improve the performance.

File Access

One other major hassle of SSI clusters is that each node has to be able to see the file systems of each other node. Why? Because if I launch a program that opens file /tmp/moshe for read/write, and then this jobs gets migrated to another node of the cluster, it has to be able to continue doing I/O to and from this file.

There were until now two options for doing so. Either the openMosix cluster intercepts all I/Os of jobs that were migrated to the current host from another node and sends those requests for I/O to that originating node.

Or, you could create a global vision of a file system through NFS. The first one is more difficult to develop, but easier to maintain in day-to-day operations. The second is easier to implement, but it could be a headache to mount all the file systems in an intelligent manner, allowing every node access to all the nodes. Additionally, you have to make sure that all userIDs and groupIDs are consistent among all the nodes in the cluster, otherwise serious permission problems will arise. And last, but certainly most important, NFS does not guarantee cache consistency for clients accessing and modifying the remote blocks. In a cache-consistent cluster like openMosix this NFS behaviour can lead to catastrophic results for applications.

Taking the best of modern file system research and applying it to openMosix, we came out with with DFSA (Direct File System Access). The DFSA file system 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.

The Implementation

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.

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 writeable 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.

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.

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/hpc. 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.