The MOSIX Multicomputer Operating System for Scalable NOW and its Dynamic Resource Sharing Algorithms *

Amnon Barak, Avner Braverman, Ilia Gilderman and Oren La'adan
Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

Abstract

MOSIX is an enhancement of UNIX that allows scalable networks of workstations and servers to share their resources. Its comprehensive resource sharing policy is geared to provide performance scalability by transparent, preemptive process migrations. This policy includes adaptive algorithms that transparently perform load-balancing, memory-sharing and IPC optimization.

1 Introduction

The increasing interest in scalable collections of platforms, including clusters of servers and Networks of Workstations (NOW) for the execution of large-scale applications, interactive and high performance computing [1], requires the development of intelligent mechanisms that can respond to resource availability, distribute the workload dynamically and utilize the available, cluster-wide resources efficiently and transparently. This is necessary for performance scalability in clusters of servers and to support a flexible use of workstations, i.e., use idle workstations whenever possible, because the overall (network wide) available resources in such systems are expected to be much larger than the available resources at any workstation or a server. We use the term NOW to refer to networks of workstations as well as clusters of distributed-memory servers on local area networks.

The main techniques for dynamic work distribution are load-balancing and load-sharing, that use some form of remote execution and process migration. These techniques can perform initial distribution of the work load among the workstations and redistribution of the work load when the system becomes unbalanced. One mechanism that can performs these tasks efficiently and transparently is preemptive process migration, which can move a process from one workstation to another dynamically. This mechanism could be supervised by various algorithms, e.g., load-balancing, in order to take advantage of available (network-wide) resources adaptively and thus to improve the performance and the overall utilization of the NOW.

In spite of the advantages of preemptive process migration, very few operating systems have implemented it for dynamic resource allocation. Instead, most existing systems provide explicit,

*Supported in part by grants from the Ministry of Defense and the Ministry of Science.
E-mail: amnon@cs.huji.ac.il, WWW: http://www.cs.huji.ac.il/mosix

user-controlled remote execution and migration, while other systems provide automated remote executions, but perform preemptive process migration at the explicit request of a user. Furthermore, very few parallel programming environments, e.g. MPVM [5], which is based on PVM [7], support process migration. For a survey of systems that provide process migration see [11].

This paper presents the NOW version of MOSIX, an enhancement of UNIX that already supports preemptive process migration, and its comprehensive policy for resource sharing. MOSIX was originally developed as a “single-system” image operating system for clusters of distributed memory, LAN and bus connected computers, where the whole cluster appears to the user's processes as one system. In order to support preemptive process migration, it was necessary to restructure the UNIX kernel to site dependent and site independent parts, to allow processes to execute in a site independent mode. These developments required a major restructuring of the UNIX kernel, and the development of several new features, e.g. a distributed file system and new communication protocols. In spite of the complexity of this project, five different version of MOSIX were developed, for different architectures and for different versions of UNIX [3].

Recently, MOSIX was redesigned for NOW configurations using the “home” model, which is similar to the “home node” of Sprite [6]. In this model all the user's processes are created and seem to run at the user's “home” workstation. In order to be able to support preemptive process migration it was necessary to enhance the UNIX kernel with mechanisms that allow the process to call the current workstation kernel directly, for local services, and to communicate with the home workstation for services that require the process environment, as defined in each user's workstation. In MOSIX, these mechanisms are implemented at the operating system kernel, without changing the UNIX interface, and they are completely transparent to the application level.

The process migration mechanism is supervised by the resource sharing policy. This policy includes three algorithms: for dynamic load-balancing, memory-sharing and optimization of the inter-process communication; and two algorithms for information gathering and dissemination. The paper presents these algorithms and explains how they inter-mesh hierarchically to form a comprehensive policy, for an efficient execution of multiple processes.

The most noticeable properties of executing applications on MOSIX are its network transparency, its preemptive process migration scheme and the flexibility of its configuration. The combined effect of these properties is that application programs do not have to know the current state of the system configuration. This is most useful for time-sharing and parallel processing systems. Users need not recompile their applications due to node or communication failures, nor be concerned about the load of the various workstations. Parallel applications can be executed by simply creating many processes, just like a single-machine environment. The operating system automatically attempts to optimize the process allocations to the availability of the (network-wide) resources. MOSIX process migration enhancements [2], shows a dramatic improvement in the NOW utilization and a significant speedup of parallel applications.

The paper is organized as follows: the next section presents MOSIX and its unique properties. Section 3 presents the process migration mechanism. Section 4 presents the overall organization of the resource sharing policy and its algorithms. Section 5 presents the load-balancing algorithms, Section 6 presents the memory-sharing algorithm and Section 7 presents the inter process communication optimization algorithms. Our conclusions and future plans are given in Section 8.

2 The MOSIX multicomputer operating system

MOSIX is an enhancement of UNIX with mechanisms for preemptive process migration, that allows transparent, efficient resource management among the (homogeneous) nodes of distributed-memory multicomputers, including LAN connected Networks of Workstations (NOW) and servers. These mechanisms respond to variations in the utilization of some resources in the workstations by migrating processes from one workstation to another, preemptively, at any stage of the life cycle of a process. The granularity of the work distribution in MOSIX is the UNIX process. Users can benefit from this execution environment by initiating multiple processes, e.g., for parallel execution. Alternatively, this operating system supports an efficient multi-user execution environment.

MOSIX is designed to run on NOW configurations that include personal workstations, file and CPU servers that are connected by LANs, a shared bus, or fast interconnection networks. A low-end configuration may include a few personal workstations that are connected by Ethernet. A larger configuration may include additional file and CPU servers that are connected by high speed ATM LANs. A high-end configuration may include a large number of nodes that are interconnected by a high performance, scalable, switch interconnect that provides a low latency and high bandwidth communication, e.g. Myrinet [4].

In MOSIX, each user interacts with the multicomputer via the user's "home" workstation. The system image model is a NOW, in which all the user's processes seem to run at the home workstation, and all the processes of each user have the execution environment of the user's workstation. Processes that migrate to other (remote) workstations use local (in the remote workstation) resources whenever possible, but interact with the user's environment through the user's workstation. As long as the requirement for resources such as the CPU or main memory, are below certain threshold levels, all the user's processes are confined to the user's workstation. When these requirements exceed the threshold levels, e.g., the load created by one CPU bound process or the size of the local memory, then the process migration mechanism (transparently) migrates one or more processes to other workstations or to the CPU servers. The overall goal is to optimize the performance by efficient utilization of the network-wide resources.

2.1 The unique properties of MOSIX

Since MOSIX is an enhancement of UNIX, it supports the UNIX interface and all its mechanisms. It also supports:

- **Network transparency** - for all cross machine operations.
- **Preemptive process migration** - that can migrate any user's process, transparently, at any time, to any available node. The migration itself involves the creation of a new process structure in the remote site, followed by a copy of the process page table and the "dirty" pages. 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 resource sharing** - that initiates process migrations to improve the overall utilization of the resources.

- **Efficient kernel communication** - that was specifically designed to reduce the overhead of the system's kernel communications, e.g. between the process and its home site, when it is executing in a remote site. The new protocol is geared for a locally distributed system. As such, it does not support general inter-networking such as routing, and it assumes a reliable media. The result is a fast, reliable datagram protocol with low startup latency and high throughput. The protocol applies a "look ahead" ACKing scheme and run-time fine tuning in order to achieve near optimal utilization of the network media and the corresponding system resources.

- **Decentralized control** - each workstation makes all its own control decisions independently and there is no master-slave relationships between the workstations. The organization of the system data bases is completely decentralized and no machine possesses information on all the system objects of any type.

- **Scaling considerations** - that ensure that the system runs as well on large configurations as it does on small configurations. The main considerations include symmetry, client/server protocols, garbage collection, and the use of randomness in the system control algorithms. Each workstation based its decisions on partial knowledge about the state of the other workstations and it does not even attempt to determine the overall state of the NOW or a specific workstation.

- **Autonomy** - each workstation is capable of operating as an independent system. This property allows a dynamic configuration, where workstations may join or leave the network with minimal disruptions.

- **File system** - MOSIX uses standard NFS.

The environment thus created allows many users to share the resources of the NOW, with maximal performance and efficient overall utilization of the available resources.

3 Process migration

The rational behind the process migration scheme is the observation that a UNIX process can be divided to two contexts: the user context, that can be migrated; and the system context, that is "home-site" dependent, and can not be migrated. We first describe these two parts of the process, then present the *Deputy*, the part of a process that can not be migrated.

3.1 The UNIX process

In MOSIX as in UNIX, a process is the basic computational unit for the execution of a program [10]. A UNIX process has two contexts, the user context and the system context. The user context contains the program code, stack and variables of the process. The system context contains description of the resources which the process is attached to, and a stack for the execution of system code on behalf of the process.

The process interacts with its environment using two mechanisms, system calls and signals. System calls enable processes to operate on resources. In order to inform the process of the occurrence of exceptional or asynchronous events, the system employs signals. Whenever such an event occurs the appropriate signal is sent to the process.

3.2 The Deputy

The main requirement for a process migration is location transparency, that is, the functional aspects of the system's behavior should not be altered as a result of migrating a process. Achieving this transparency requires that the system is able to locate the process, e.g. in order to preserve the user's interface; and that the process is unaware of the fact that it has been moved from one node to another. A simple way to achieve these two requirements is to maintain in the "home" site of the process a structure that represents the process and interacts with the process environment. In MOSIX, the "home" site of the process is the user's workstation. The structure that represents the process in that site is called the *Deputy*.

The concept of the *Deputy* of a process is based on the observation that only the system context of a process is site dependent. Generally the idea is that the user context of a process is migrateable, and since its interface to the system context is well defined, it is possible to intercept every interaction between the user and system contexts, and forward this interaction across the network.

In order to be able to migrate a process, it was divided to two parts: the *Body* which has all of the user context and almost none of the system context of the process, and the *Deputy* which has the rest of the system context, and no user context. The user context and the part of the system context that is associated with the *Body* are site independent. The *Body* can, therefore, be migrated to any node. The *Deputy* has the site dependent part of the system context of the process, hence it must remain in the home site of the process. The two parts are connected by a communication channel, on which interaction between the two parts takes place. Figure 1 shows two processes that share a home site. One process is unsplit (process A) and the other is a split process (process B) that has been migrated.

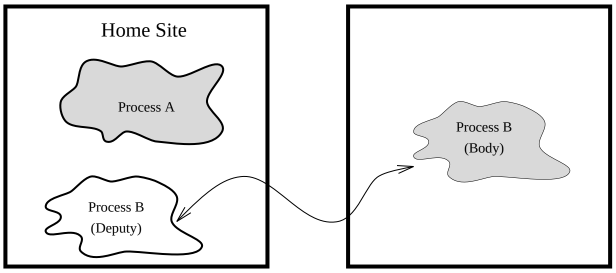

Figure 1: *Unsplit, and split processes*

In the execution of a process in MOSIX, location transparency is achieved by forwarding site dependent system call to the home site. System calls that are executed by the *Body* are intercepted by the remote site's kernel. If the system call is site independent it is executed locally, otherwise, the system call is forwarded to the *Deputy*. The *Deputy* executes the system call on behalf of the

Body in the home site, and sends the result back to the *Body*, which then continues to execute the user's code.

The home site *Deputy* approach offers a simple solution to the process location requirement. Instead of providing the location of the process, the *Deputy* provides a channel to interact with the *Body*. The kernel at the home site informs the *Deputy* of asynchronous events, the *Deputy* checks if there is any action to be taken, and informs the *Body* if so. The *Body* checks the communication channel for reports of asynchronous events in the same locations a standard UNIX process checks for signals. The *Deputy* approach is also a very appealing method to achieve a clean solution to the transparency requirement. This solution is robust, and is not affected even by major modifications of the UNIX kernel. Some modifications, such as supporting a distributed file system, might even improve the solution by making many system calls site independent. Also, this solution seems to be portable, it relies on almost no machine dependent features of the kernel, and thus does not hinder its porting to different architectures. In fact, it does not rely much on UNIX specific properties, and seems to be exportable to other time-sharing operating systems.

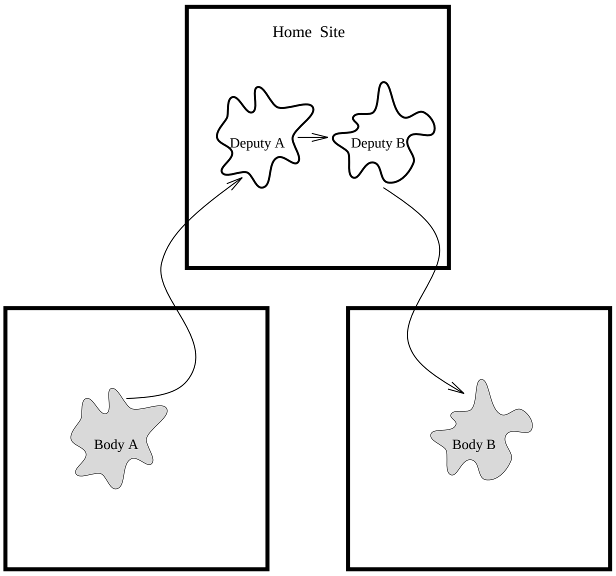

Figure 2: *The path of data sent from process A to process B*

The *Deputy* approach has, however, some overhead in the execution of system calls. For example, as Figure 2 shows, data sent from one process to another has two hops over the network, instead of just one. This implies some scaling limitations due to a bottleneck at the home site, when executing a large number of communicating processes. Additional overhead is incurred due to the need to hold enough state at the home site, and to keep a dedicated communication channel.

3.3 Process migration in a fast LAN

The process migration mechanism is a key feature for resource sharing in a NOW. It has a significant influence on issues such as system stability, resource utilization and the overall performance. Therefore, the overhead incurred by such a mechanism, must be minimized. Typically, process migration consists of two stages. In the first stage two workstations negotiate and exchange initial data, using relatively small ping-pong messages. The time overhead for this stage is mostly due to the network latency. In the second stage the process' dirty pages are transferred, and the network bandwidth prevails.

One would expect, therefore, that the migration cost will depend mostly on the underlying network characteristics. Hence, a Gigabit-per-second network, e.g. Myrinet, should yield a migration "in the speed of light". Unfortunately achieving high utilization from very fast networks is a complicated task. The reason for this lies in the design of the operating system networking software. When the first protocol stack was developed (about 20 years ago) the data transfer rate between the processor and the network was 1000 times larger than the network transfer rate. Therefore, there wasn't a real need for an efficient design and implementation. Today, fast networks may be 2-4 times faster than the I/O bus, and they are essentially underutilized. The throughput is further reduced due to inefficient design of the networking system.

Consider, for example the TCP/IP protocol. This protocol was designed along the scheme of the OSI 7 layers model. The data passes several layers starting from the user level, down to the wire, and each layer incurs some overhead. Typically this overhead is due to data copy, which is a costly operation in the context of high speed networking. The first layer is the *socket layer*, which provides basic data flow control and buffering services. The data is copied from the user space into the kernel space. The second layer is the transport protocol TCP. Since it is a reliable protocol, it keeps a copy of the data (second copy operation) for a possible retransmission. TCP also calculates a checksum of the data, which implies a third pass over the data. The data proceeds to the IP layer, which incurs further overhead due to fragmentation. Lastly, the data reaches the device driver, which copies the data into the network interface (either using programmed I/O or a DMA engine). The above scheme yields a typical total of 4 data copy operations before the first byte appears on the wire. The net effect is a dramatic degradation in the network bandwidth.

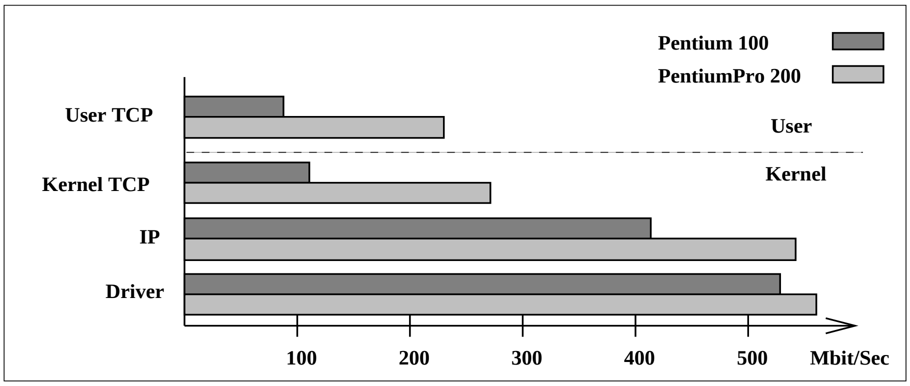

Figure 3: *TCP/IP performance breakdown for the Myrinet LAN*

Figure 3 presents the performance breakdown of the TCP/IP protocol for the Myrinet LAN, on a Pentium 100 and a Pentium Pro 200. The figure presents the cumulative bandwidth performance of four different communication layers. This partitioning demonstrates the performance degradation of each layer. The bottom layer presents the maximal transfer rates between the machine and the LAN interface. In the Myrinet, this is done by the DMA engine, which is independent of the processor speed. This implies nearly identical performance for the two processors. The next layer presents the performance of the IP protocol. The difference in the performance between the two processors is due to the (de)fragmentation of the data. The kernel-TCP and the user-TCP layers represent the throughput of the entire protocol stack from within the kernel and by the user level respectively. The difference between these two cases is the overhead of entering the kernel. There is a significant slowdown of these layers vs. the IP layer due to 2-3 data passing (copy) operations. These operations strongly depend on the speed of the processor. Thus the Pentium Pro-200 outperforms the three times slower Pentium-100 by a factor of 2.7.

The above discussion implies that future networking software will require a substantial reduction in the number of the data copy operations. This can be achieved by the following measures:

- **Efficient buffer management** - instead of physically copying the data, a special field can designate the reference count of the data. Since the layers do not change the data, but only prepend or append new data, we only need to be able to save the *structure* of the data. Using this information, we can later decrease the reference count and recover the data.

  Hardware checksum/CRC services - most modern network interfaces already offer data integrity testing tools such as checksum and/or CRC calculation. Since such services were not available in the hardware in the past, they were done in software.

  Scatter/gather operations - combining these facilities with an efficient DMA engine saves the system the burden of copying the data to/from a contiguous buffer. Such facilities are nowadays common to many network interfaces. Note that the DMA engine is faster than programmed I/O for bulk data transfers, and it incurs minimal overhead.

  Memory mapping - sometimes the application knows that the data will not be modified until its reception at the peer. In such a case, the data can be mapped to the network buffers (rather than copied), and accessed directly by the DMA engine. Note that the process migration mechanism meets the above criterion: a migrating process stops executing when the migration begins, and resumes its execution only when it arrives safely at its destination.

3.4 Performance of the process migration mechanism

Process migration is carried out in two stages. The first stage involves establishing a connection and negotiations between two workstations and creating a process frame in the remote workstation. In the second stage the active memory of the process is transferred. The cost of the negotiations stage is fixed, while the data transfer cost is proportional to the amount of data transfered. Thus the migration time is given by:

$$Migration\ time\ =\ Basic\ cost + Cost\ per\ page \times Process\ size$$

The basic costs and the cost per page for Ethernet-100 and the Myrinet LANs between two Pentium Pro 200MHz workstations are shown in Figure4. From the figure it can be seen that the migration time is a linear function of the process size. It amounts to 76.8Mb/s for Ethernet-100 and 228.8Mb/s for the Myrinet. The corresponding migration latencies (basic costs) are magnified in the figure. They are 3ms and 2ms respectively.

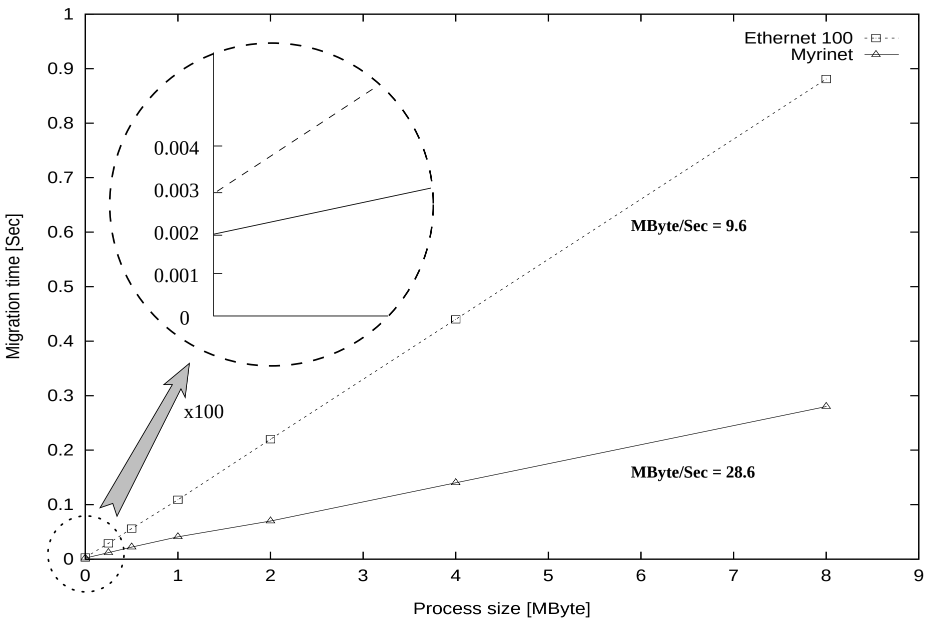

Figure 4: *Migration times vs. process size*

4 The resource sharing policy

This section presents the overall organization of the resource sharing policy and its algorithms. This policy includes three algorithms for resource handling (sharing and allocation) and two algorithms for information gathering and dissemination. The resource handling algorithms use the (preemptive) process migration mechanism for load-balancing, memory-sharing and for optimization of the Inter-Process Communication (IPC). In order to reduce the overhead, all the algorithms are executed in a decentralized manner and they are not synchronized, i.e. each workstation executes all the algorithms independently. Furthermore, due to scaling considerations and to further reduce the overhead, each workstation uses partial knowledge about the state of the other workstations and it does not even attempt to know the overall state of the system or a specific workstation.

The first and most dominant algorithm is the load-balancing algorithm. This algorithm prevails as long as there are no extreme shortage of resources, such as memory or empty slots in the system tables. The load-balancing algorithm is described in the next section.

The memory-sharing algorithm is triggered when a workstation starts excessive paging due to insufficient free memory for a new process. In this case the algorithm overrides the load-balancing algorithm and it attempts to migrate a process to a workstation which has sufficient free memory, even if this migration may results in an uneven load distribution. We note that we use the term "memory-sharing" to describe the use of remote memory resources to avoid local paging and thus provide processes with the memory resources they require. This parallels load-sharing where the goal is to provide a process with the CPU resources it needs. The memory-sharing algorithm is described in Section 6.

The IPC optimization algorithm attempts to reduce the communication overhead by migrating pairs of communicating processes to the same workstation, to benefit from fast shared memory communication. The decisions made by this algorithms are subject to the load of the workstations and the relative load implied by each of the processes involved. The IPC optimization algorithm is described in Section 7.

4.1 The information gathering and dissemination algorithms

The algorithms for information gathering and dissemination are geared to provide each workstation with sufficient information about available resources in other workstations, in order to allow each workstation to make its resource sharing decisions independently, without polling or further reliance on remote information that is not already available locally.

The information gathering algorithms measure the amounts of the available resources at each workstation. To dampen momentary fluctuations, each algorithm is executed several times each second. A key component of these algorithms is the choice of suitable resource indices, which reflects the availability of the local resources to possible incoming processes from other workstations. For example, to be effective, the load index should correlate well with the task response time. The specific algorithms are:

- **The load index algorithm** - measures the average ready-to-run queue of each workstation and then correlate this number with the speed of the workstation. The rational behind this choice is to reflect as much as possible the availability of the CPU to ready-to-run processes.
- **The free memory index algorithm** - measures the amount of the workstation's free memory that may be allocated to remote processes. This is the sum of the actual free memory less the free memory threshold value (see below), and the amount of memory already committed to arriving processes. The free memory index is measured several times a second so as to dampen transient fluctuations.
- **The communication index algorithm** - determines the remote node(s) to which the current node performs most of its communication. The specific amounts are decreased each unit of time in order to dampen old values.

In MOSIX, these indices form the workstation's Combined Resource Index (CRI), which is then sent at regular intervals to other workstations, by the information dissemination algorithm. This algorithm consists of two parts, the sender and the receiver. The sender chooses a random workstation ID number (within the range of the participating workstations) and then it sends the most recent CRI to that workstation. The message transmission does not require an acknowledgment and the sending site does not know (or care) about the status of the receiver site. We note that the use of random workstation ID is due to scaling considerations, for even distribution of CRI's among the participating workstations, to support a dynamic configuration and to overcome partial (workstations) failures.

The receiver algorithm maintains a (small) structure (window), with the values of the most recently arrived CRI values. The receiver algorithm is responsible to place new values in that window and flush out older values. An interesting question is what should be the size of this (sliding) window. On one hand, too big of a window implies that it has old CRI values that might be outdated. On the other hand, if the window is too small, then the local workstation might not have sufficient information to make good decisions. Simulations of this problem show that a relatively small window, e.g. of size $logN$, where N is the number of workstations, is sufficient for good information dissemination.

5 The load balancing algorithms

The most dominant part of the resource management policy are the load-balancing algorithms. These algorithms prevails as long as there is no extreme shortage of resources such as free memory. The goal of the algorithms is to balance the loads of all the participating workstations. This is accomplished by continuous attempts to reduce the load differences between pairs of workstations by migrating processes from overloaded workstations to under utilized workstations. This scheme is decentralized in the sense that all of the workstations execute the same algorithms, and that the reduction of the load differences is performed independently by pairs of workstations. Other goals of the load-balancing algorithms are to respond to changes in the loads of the workstations, the runtime characteristics of the processes, and the number of workstations.

The load-balancing algorithms are designed to respond quickly to fluctuations in the workstation's loads and other events that may cause uneven resource usage. At the same time, the algorithms use several threshold functions to stabilize the load by avoiding excessive migration and useless oscillating of processes from one workstation to another. A simple mechanism to impose stability is to require a process to establish residency at a site by executing for a minimum period before it is considered eligible for migration. One advantage of this requirement is that short-lived processes are not migrated.

An important parameter of the load-balancing algorithms is the length of the residency period. The goal is to avoid useless migrations by migrating a process only if it results in some performance gain, e.g., a reduction in the process's execution time. Since the remaining execution time of a process is unknown, no method can guarantee that performance will be improved, or that it will not suffer from the migration. In practice, it is generally assumed that processes will exhibit a "past-repeat" execution pattern; that is, a process is expected to continue its execution for a period of time that is at least equal to the time it has executed so far. See [8] for further discussion of this issue. Based on this probabilistic assumption, and since the migration itself requires time, CPU cycles, and communication, the residency period is determined by a competitive algorithm. Briefly, this competitive algorithm allows a process to migrate only after it executes for a period of time that equals the predicted migration time. Note that this duration depends on the speed and current load of the workstations and the network. In spite of these, a migration policy that is based on competitive algorithms is simple to implement and provides an adequate method for imposing the residency requirement, and also adapts well to the current state of the participating workstations.

The main algorithms of the load-balancing scheme are: *when* to migrate a process, *where* to migrate it and *which* process to migrate [3].

- **When:** this algorithm is activated whenever the difference between the local load and the load of another workstation (whose load value is present in the local information window), is above a threshold value, e.g., the load created by a CPU bound process.

- **Where:** the choice of the specific target workstation depends on the availability of load index values of some subset of remote workstations, the relative speeds of these workstations, and the communication overheads. Usually, the workstation with the lowest load is chosen, unless it does not have sufficient memory to accommodate the migrating process.
- **Which:** the decision of which process to migrate depends on the measured execution profile of each process, its use of (local) resources, and its accumulated execution time. The current implementation gives higher priority for migration to CPU-bound processes with the longest accumulated execution time.

5.1 Performance of the load balancing algorithms

In order to measure the efficiency of the load-balancing algorithms, we executed a set of identical CPU-bound processes that were created simultaneously in one workstation. The measured execution times are compared to the theoretical execution times of the same processes using the optimal static assignment. The execution platform is a NOW configuration with 16 identical (Pentium based) workstations that are connected by an Ethernet.

No. of Procs.	Optimal Time	MOSIX Time	Slow-down(%)
1	300	301.91	0.6
2	300	302.92	1.0
4	300	304.57	1.5
8	300	305.73	1.9
16	300	310.83	3.6
17	450	456.91	1.5
20	450	462.07	2.7
24	450	471.87	4.9
31	563	574.03	1.9
32	600	603.17	0.5
33	700	705.93	0.8
36	700	715.35	2.2
38	750	759.90	1.3
40	750	767.67	2.4
48	900	916.11	1.9

Table 1: Optimal vs. MOSIX load balancing performance

Table 1 presents the results of this benchmark. The execution time of each process was set to 300 seconds. The first column lists the number of processes. The second column lists the theoretical execution times (in seconds) of these processes, assuming no communication overhead and a preemptive process migration when the difference between the loads of two nodes is greater than the load created by one CPU bound process. Column three lists the measured execution times (in seconds) of the processes using the MOSIX load balancing algorithms, and column four the slowdown ratio (in %) between column three and column two.

From Table 1 it follows that the average slow-down ratio of the load balancing algorithm vs. the optimal execution algorithm is 1.95%. We note that the MOSIX imposes a minimal residency (local execution time) of 1 Second. This residency time is the main reason for the difference between the results of column two and column three in Table 1.

6 The memory sharing algorithm

The goal of the memory-sharing algorithm is to place the maximal number of active (ready to run) processes in "network memory" across all the workstations. It attempts to avoid processes thrashing or the swapping out of processes, even if this causes the loads in the nodes to become unbalanced. Although a process is not active whilst being migrated, it is able to resume execution as soon as it arrives at the destination workstation. In contrast, a process that is swapped out is suspended. Another drawback is when a process starts paging, it imposes a severe slowdown on all the executing processes in that workstation. Process migration, on the other hand, reduces the load and thus improves the execution environment of the remaining processes.

The memory-sharing algorithm is most useful in cases where many workstations are evenly loaded, but memory is not being used uniformly. It is also useful in cases when a user suddenly creates several processes that consume all the available memory in the home workstation. In order to reach a decision, the algorithm needs to decide *when* to migrate a process, *where* to place it and *which* process to migrate. In addition to these, the algorithm needs to decide whether to allow remote processes to move in due to memory depletion in another workstation. The general policy is that only if the workstation has the resources, e.g. sufficient free memory, then processes are allowed to move in.

- **When:** this algorithm is triggered when a workstation's free memory falls below a threshold value that may cause valuable pages to be paged-out. This threshold value is an area of memory reserved for the use of local processes whilst the decision to choose and then migrate away a process is made.

- **Where:** the target workstation is chosen from the subset of workstations whose resource indices are currently available locally, and whose free memory indices are greater than or equal to the size of the process designated for migration. From these, the workstation with the lowest load is normally chosen. This is to avoid, as much as possible, further process migrations due to load imbalances after the current migration.

- **Which:** initially, the process with the lowest migration overhead is selected from the processes whose size (in-core dirty pages) is sufficiently large to bring the free memory size bellow the threshold value. If this fails, then the algorithm attempts to migrate the largest possible process that fits into a remote workstation without causing that (target) workstation to exceed its free memory threshold value. Afterwards, if the available free memory is still under the threshold value, then the algorithm is executed again.

6.1 Performance of the memory sharing algorithm

This test presents cases where the initial load of the workstations is balanced, and the workstations have different amounts of free memory. Assume that a new set of processes is created, and that the processes are evenly distributed among the workstations, e.g. as in PVM, such that some workstations run out of free memory. In this case, there is no justification for process migration due to load-balancing.

The benchmark included sets of 32 CPU-bound processes that were assigned in a round-robin fashion to 8 workstations, each with 21MB free memory. In 7 workstations the active memory size of each process was set to 4MB, totaling 16MB in each of these workstations. In the eighth workstation the size of the processes was increased in each execution from 4 to 14MB (totaling 16 to 56MB respectively). Note that for the last case, i.e., process size 14MB, the total size of all the processes was equal to the total network memory.

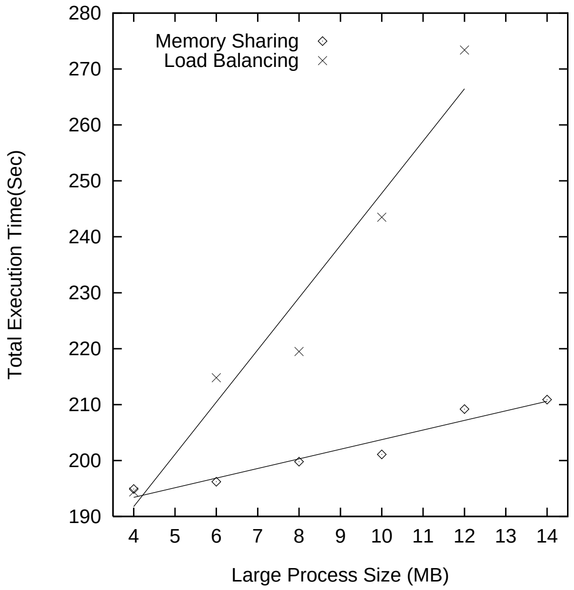

Figure 5: *Performance of the memory-sharing algorithm*

The benchmark was executed in a load-balancing system with and without the memory-sharing algorithm. The results of the executions, with linear best-fits, are shown in Figure 5. As can be seen from the figure, when the total requirement for memory is increased, the execution times of the load-balancing algorithm without memory-sharing are sharply increased due to an increased paging activities. The corresponding execution times with the memory-sharing have a significantly lower increase due to much lower paging activities.

7 The IPC optimization algorithm

The goal of the Inter-Process Communications (IPC) optimization algorithm is to minimize the communication overhead of distributed applications, while keeping the workstations load balanced.

These two requirements are essentially contradictory by their nature. Load balancing is achieved by distributing the processes evenly, to utilize all the workstations. On the other hand, the communication overhead is minimized if clusters of communicating processes are assigned to the same workstation, to benefit from fast shared-memory IPC. As a result, there are scenarios where load balancing is the dominant consideration, e.g., low communication volume, while in other circumstances communication optimization prevails. The above observations lead to the conclusion that a combined approach for load-balancing and IPC optimization should be used, rather than different algorithms for each case. This section describes one such combined algorithm.

The design approach of the algorithm is based on a commodity market model. This model consists of a set of producers, that offer products at given costs, and a set of consumers, that are in need of these products. As in a real market, the consumers are looking for producers which offer a required product for the lowest price. In the NOW environment, the workstations are producers and the products are the various resources, e.g., CPU cycles, communication links, memory, I/O devices, etc. The consumers are naturally the processes, which need the resources for their execution. Normally a resource is available on more than a single workstation, and will be offered for different costs. We define the cost of a resource on a certain workstation as the amount of time that is needed to use one unit of that resource. Hence, the higher the cost of the resources used by a process, the larger is its execution time. In order to minimize the response time of a process, it should migrate to the workstation that offers the lowest price for its particular needs.

Consider the following examples of common resources. First consider the resource of *CPU-cycle*. The cost of this resource on a workstation, is proportional to the load on that workstation. A process seeking the lowest *CPU-cycle* cost will therefore choose to migrate to the least loaded workstation. Another important resource is the *communication links*. The cost of a *communication link* of a process (to another process) is negligible if they both execute on the same workstation. However, if they execute on separate workstations, then the cost becomes the time to transfer the data over the network. Hence, a process will prefer to join its peer, purchase the *communication link* resource at the a lower cost and thus lessen its execution time. Lastly, consider the *memory* resource. The cost of this resource on a certain machine is zero, as long as there is enough free memory. When the memory becomes scarce, the price becomes very high, i.e., a step function. We note that from the above examples it follows that the cost of a resource is a function of both the workstation (producer) and the process requiring it (consumer). Thus the costs are defined by three-tuples of the form $<$*resource,workstation,process*$>$. We term this set of tuples the "price-list".

The algorithm employs the preemptive process migration mechanism to migrate processes to workstations where they can purchase their resources for the lowest price. In order to decide on such migrations (the when, which and where questions), the algorithm continuously calculates the resource demand profile of the processes, such as CPU usage, communication patterns and rates, memory usage etc. In addition, information about resource usage and availability in the various workstations is disseminated in the NOW. Only then, based on these values, can the price-list be derived. The algorithm uses these values to calculate the **execution cost indices** of the processes. An **execution cost index** of a process-workstation pair estimates the response time of the process on that workstation. This value is essentially the total cost that the process would pay on that workstation. Therefore, a migration decision is reached using these indices according to the following three criteria:

- **When:** a migration is triggered when there exists a process whose local execution cost index is larger than its execution cost index on another workstation. Note that this difference must be larger than the migration cost of the process.
- **Which:** the migrant is chosen to be the process that has the largest expected gain due to the migration. In other words, we choose the process with the maximal difference between its local index and an index on another workstation.
- **Where:** the target workstation is naturally the one for which the process currently has the minimal execution cost index.

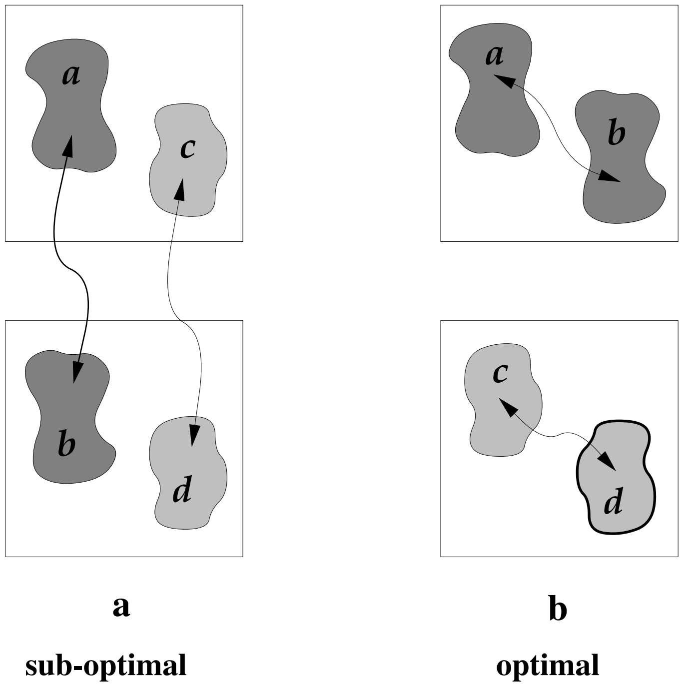

Figure 6: *Process exchange migration: exchanging processes C and B reduces the communication overhead.*

The algorithm repeatedly applies the above policy to reduce the process' response time. We term such migration decisions "single-tone" decisions, because they are based on considerations of a single process. However, there are many cases for which single-tone decision are not sufficient. Consider, for example, the case of two pairs of identical communicating processes, that were initially been assigned to two workstations, as shown in Figure 6. If the assignment is such that each communicating pair is assigned to different workstations, as shown in Figure 6a, then this results in a higher communication overhead than an assignment that places each communicating pair in the same workstation, as shown in Figure 6b, in which case the processes benefit from fast local communication. Clearly, the second alternative is better than the first. However, if the assignment was already done using the first case, then there is no incentive for any single process to migrate since this may cause the system to become imbalanced. On the other hand, if two processes are exchanged between the workstations, then the communication overhead is optimized and the load of the workstations remain balanced. The above *process exchange* operation requires both parties to negotiate and coordinate the migrations. Therefore we classify it a "dual-tone" decision.

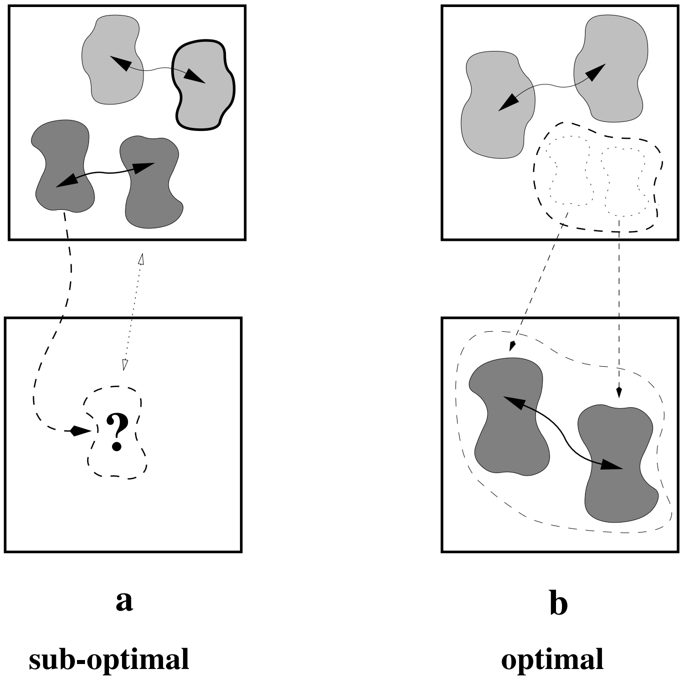

Figure 7: *Double process migration: a single process would not migrate (a), but 2 communicating processes can migrate together to reduce the load (b).*

Another dual-tone case is when two communicating processes execute on a loaded workstation, as shown in Figure 7a. In this case none of the processes will migrate to a less loaded workstation, because this will incur a high IPC cost. However, if the two processes are migrated together (as shown in Figure 7b), i.e. a *double migration*, they will execute on a less loaded workstation, while benefiting from a low IPC overhead. A third dual-tone operation, called *processes merge*, migrates two communicating processes from two different workstations to a third workstation.

The above discussion can be used to define a set of *atomic migration operations*. This includes regular (single-tone) migration, and three dual-tone migrations, including process exchange, double migration and merge migrations.

The algorithm uses these operations in order to repeatedly bring the system to an optimized configuration. More complex operations can be build by composing several atomic operations.

8 Conclusions

This paper presented an overview of MOSIX and its comprehensive resource sharing policy. This policy consists of algorithms for load-balancing, memory-sharing and IPC optimization. The algorithms are executed independently by each participating workstation, they are decentralized and inter-mesh hierarchically with other algorithms for information gathering and dissemination, where each algorithm serves as the foundation for the next level algorithm. Together, these algorithms create an efficient execution environment, with maximal performance and high degree of overall resource utilization in a scalable NOW.

In the described resource sharing policy, each workstation based its decisions on partial knowledge about the state of the other workstations and it does not attempt to determine the overall state of the NOW or any specific workstation. This property ensures that the system runs as well on large configurations as it does on small configurations.

The combination of process migration with the newly emerging low latency and high bandwidth communication networks, such as the Myrinet [4] LAN, opens new dimensions for the execution of large-scale applications, interactive and high performance computing. For example, it would be interesting to develop algorithms to utilize available remote network-memory. These algorithms should adapt to the network load and the availability of the remote memory. It would also be interesting to develop a *memory server* that can swap portions of a large program to idle memory in remote workstations. Both algorithms could benefit from our process migration mechanism, which has the capability to page across the network. These project are similar to the network RAM project described in [1]. Another possibility is to use the process migration mechanism to support shared memory, by bringing the process to the data rather than the data to the process. The advantage of this approach is that it scales up without jeopardizing the consistency of the data due to many writers, since all such processes would be migrated to the same workstation.

The NOW MOSIX operating system is compatible with BSD/OS [9], which is based on Release 4.4BSD-Lite from the Computer Systems Research Group at UC Berkeley. The current implementation has been operational for over 3 years on a large cluster of Pentium based and several i486 based workstations. This system is used for student course work, research and development of operating systems for scalable multicomputers, and the development of parallel applications. Its unique mechanisms provide a convenient environment for writing and executing parallel programs with minimal burden to the application programmers.

A limited (up to 6 processors) version of MOSIX, called MO6, is available on the Internet at URL `http://www.cs.huji.ac.il/mosix`. MO6 allows users of BSD/OS to build a powerful, low-cost, distributed memory multicomputer.

Acknowledgments

We would like to thank Amnon Shiloh and Yuval Yarom for their contributions.

References

[1] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] 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.

[3] 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.

[4] 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, February 1995.

[5] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[6] F. Douglis and J. Ousterhout. Transparent Process Miration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM - Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[8] M. Harchol-Balter and A.B. Downey. Exloiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proc. ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*, pages 13–24, May 1996.

[9] R. Kolstad, T. Sanders, J. Polk, and M. Karles. *BSDI Internet Server (BSD/OS 2.1) Release Notes.* Berkeley Software Design, Inc., Colorado Springs, CO, January 1996.

[10] M.K. McKusick, M.J. Karels, K. Bostic, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, MA, 1996.

[11] M. Nuttall. Survey of Systems Providing Process or Object Migration. Technical Report DoC 94/10, Imperial College, London, May 1994.