Load Balancing Experiments in openMOSIX

J. Michael Meehan
Computer Science Department
Western Washington University
Bellingham, Washington, 86225
meehan@wwu.edu

Adam Wynne
Western Washington University
Bellingham, Washington,986225
wynnea@cc.wwu.edu

ABSTRACT

This work concerns the study of load balancing in the openMosix distributed operating system. The openMosix kernel was augmented with the means to gather performance data related to the operation of its load balancing policies. A performance baseline was established via a number of tests. The load balancing information exchange algorithm was modified in various ways. This study focuses on attempting to determine the correct size for the load vector used to store and exchange load information amongst the nodes. We present the conclusions drawn from these studies and suggestions for future work.

1 INTRODUCTION

Clusters of inexpensive networked computers are increasingly a popular platform for executing computationally intense and long running applications or simply for systems requiring extremely high levels of throughput. The common practice has been to use application controlled *remote execution* to take advantage of cluster resources. Under this approach the system implementer must design into each application the logic required to determine an appropriate distribution of tasks in order to reap the benefits of the multi-computer. It has long been thought that eventually such "hand tailored" distributed applications would be replaced by distributed operating systems. In a distributed operating system the various nodes communicate with each other in order to produce a single collective resource from the point of view of the users and applications. In order to produce such an operating system one of the crucial issues is how the various nodes exchange load information in order to make informed decisions concerning the migration of processes from one node to another. The policies concerning load balancing must be efficiently and effectively implemented striking a balance between message overhead and effectiveness. This has been an area of a considerable amount of research over the last twenty five years. Some of the more well know distributed operating system projects prior to the year 2000 include Sprite [14], V [5], LOCUS [15] Amoeba [16] and MOSIX [4]. Distributed operating systems projects still active or initiated post 2000 include Concert [1], Kerrighed [11], GENESIS [8], MOSIX and openMosix [3,13].[1]

1.1 BACKGROUND - MOSIX to openMOSIX

This work utilizes the readily available open source distributed operating system *openMOSIX*. The openMosix project [3,11] is derived from MOSIX and is released under an open source software license. MOSIX has a long history having been implemented on top of various versions of UNIX. The current version of MOSIX exists as a patch to a Linux kernel. At the time the work described here was carried out the stable version of openMOSIX was a patch to the 2.4 Linux kernel. The 2.6 Linux kernel openMOSIX patch was in the process of being developed. OpenMOSIX uses local resources to satisfy SVC calls whenever possible, but often system calls must be redirected to a process' UHN. This is done by utilizing a *deputy process.* A deputy is a remotely executing process' representative at its UHN[9].

2 METHODOLOGY

In order to investigate the performance of modifications to the standard openMOSIX load balancing implementation it was first necessary to augment the kernel with performance instrumentation code. We were very sensitive to the fact that any modifications to the kernel to put this instrumentation in place could have the effect of changing the behavior of what we were trying to measure. The data gathering instruments have been implemented as data loggers which write to a special character device implemented in a kernel module. This approach has the added benefit of making it easy to retrieve and process the data collected by simply reading from this character device after an experiment has been conducted. An entry in the /proc file system was created to control switching on and off data collection. In order to inject loads into the cluster in a controlled and repeatable manner. a test scripting tool was constructed.

The hardware used to conduct the experiments was a cluster

1 For an overview of these operating systems and other material related to distributed operating systems see http://faculty.cs.wwu.edu/DistOs

of twenty identical machines. Each machine was a 2.4 GHz Celeron with 256 MB of RAM. The machines were all located on the same Ethernet switch. The network connection speed for all machines was 1 Giga-bit/second.

3 LOAD BALANCING IN OPENMOSIX

openMosix uses a probabilistic, decentralized approach to disseminate load balancing information to other nodes in the cluster. This allows load information to be distributed efficiently while allowing excellent scalability. The major components of the openMosix load balancing scheme are the information dissemination and migration kernel daemons. The information dissemination daemon runs on each node and is responsible for sending and receiving load messages to / from other nodes. The migration daemon also receives migration requests from other nodes and if willing carries out the migration of the process. In addition, a third daemon, the memory daemon, is started at system initialization to obtain an ongoing profile of memory utilization.

The information daemon has two basic functions, the sending and receiving of load information. The sending part operates as follows. A periodic alarm, set to 1 second in the current implementation, is used to trigger the sending of load information. Two nodes are selected at random from two groups. The first group consists of all nodes that have contacted us "recently" with their load information. The second group is chosen from all nodes in the cluster.[2] These two nodes are then informed of the sending node's current load.

The load calculation is attached to the clock interrupt routine. At each clock interrupt the number of running processes is added to an accumulator. When the number of clock interrupts completes a second the load is calculated. The accumulated load is first normalized to the CPU speed of this processor using the maximum CPU speed in the system versus the local node's calculated CPU speed.[3] In our cluster all the machines have identical speeds and thus this is irrelevant. The load is computed as a combination of the old load modified by a decay value added to the newly accumulated load number. The system then compares the accumulated load for the last second with a value representing the highest load it has recorded over any one second period. The maximum of the two is retained. If the current load is smaller than the current maximum load the new maximum load is computed to be the old maximum load times 7 plus the newly accumulated load all divided by 8. In this fashion, the maximum observed load decays away gradually if the current load continues to be smaller than the maximum. The load information sent to the two randomly selected nodes is different from this load calculated for internal use. This is part of the strategy used to prevent migration thrashing. The actual load value sent to other nodes, known as the export load, is slightly higher than the internally calculated load. The export load value is a combination of the decaying maximum load value described above and a stabilizing factor. In addition, the export load is increased by the CPU utilized in the last second by any process "recently" migrated to this node. This is done to prevent receiving too many new processes too quickly. Finally a value is added which increases with the number of processes currently in the process table to make nodes with nearly full process tables less attractive than those with fewer processes.

The receiving portion of the information dissemination daemon performs the following actions. Upon receiving load information from another node the information is added to the local load vector. The standard implementation simply utilizes a circular queue of eight entries. Thus, the oldest information is overwritten by newly received information. After the new load information has been placed in the queue a check is made to determine if there exists a node in the eight entries that has a lower load than this node. A target node for the migration is NOT selected at this point. Instead the *choose()* function is executed to determine which process will be selected for migration. Determination of the process to be migrated is based on the following criteria.

Processes can be locked, meaning they are not subject to migration. If a process is still in its creation phase it is not considered. If a process is in the act of being transferred to another node it is obviously not considered. If a process is using a shared memory region it is not considered. [4] If any

2 The list of nodes comes from a configuration file. It is possible to select a node which is not up. openMOSIX can also utilize an auto discovery option in which case only nodes that have been in operation since this node booted will be added to the list. It is still possible to select a node which is not currently up.

3 The maximum CPU value is settable by the operator by placing a value in a file in /proc. It would be better it at initialization a node exchange values with each other node.

4 This is likely to be changed in the next release.

process is found which has already been selected for migration the search is aborted. If a process has not accumulated enough CPU usage to reach residency on this node then it is not considered for migration. The residency period is set to an estimate of the amount of time it would take to migrate the process over the network. There is a threshold value which must be met for a process to be considered for migration. The attractiveness of a process as a candidate is based on its CPU use since we last visited its PCB looking for candidates. This is combined with a value which attempts to measure the process' contribution to the load currently on the machine. Processes which are "frequent forkers" are given an offset to make them more attractive for migration under the reasoning the once migrated they will continue to fork children thus spreading the load as they are bounced from one node to another.

Once the process to be migrated has been selected a flag is set in its PCB.. Just before placing a process into execution the dispatcher looks to see if the process is to be migrated. If it is, the *consider()* function is executed. The consider function runs in its own kernel thread and the dispatcher goes about its normal routine. The *consider()* function determines the target node for the migration by computing the opportunity cost for each of the eight nodes in the local load vector. For an explanation of the concept of an opportunity cost for migration target selection refer to the paper by Yair Amir et al. [2] For a simplified explanation of the way the opportunity cost is computed in openMOSIX consider the sum (current cpu usage / maximum cpu usage) + (current memory usage / maximum memory usage) for each node in the load vector. The "marginal cost" is the amount this sum would increase if the process was migrated to that node. The idea is that minimizing this marginal cost provides the optimal target node for migration. The node with the least opportunity cost is selected. An attempt is made to migrate the selected process to this node. The prospective receiving node may decline the migration in which case the next node in an array sorted by opportunity cost is tried. The local node is usually included in the target selection process. The local node is made less attractive by adding in offsets for situations such as nearly out of memory or the process table is nearly full etc. If the local node is selected then the process is simply "unmarked" for migration.

3.1 Observations

You will observe that whenever new load information arrives from another node, finding a node in the load vector (8 nodes in the standard kernel) with a load less than this node triggers the search of *all* process control blocks in the system. It is possible to do all the work to search all the PCBs to select a process and then not have anywhere to migrate it. It is also possible that the local node ends up with the least opportunity cost and so the chosen process is not migrated. In this case all the time spent in the *choose()* and *consider()* functions was for naught. In addition, all nodes selected as migration targets could refuse to accept the migration, although this is not likely.

It is import to understand that the system does not directly incorporate a lower bound for the difference between our load and another node's load. This is handled by each node advertising a slightly modified version of its load rather than the value computed for internal use, the internal load versus the export load. Thus, there is in a sense a built-in offset between our load value and the numbers found in the load vector. Each machine believes each other machine is slightly busier than it really is. This built-in offset accomplishes two things. It prevents the long search of all process control block in the event that the difference between the local node and the remote nodes in negligible. In addition, it prevents, to some degree, migration thrashing which would occur if no retarding factor were applied.

3.2 Data collected in the experiments

3.2.1 Counters

1.Number of load messages received from other nodes [hpc/info.c: info_recv_message()].

2.The number of times choose() is run. This represents the number of times we decide to migrate a process [hpc/balance.c: load_balance()].The number of processes migrated *to* this node [hpc/mig.c:mig_remote_receive_pro()]

3.The number of times we receive a GETLOAD message. i.e., another node is asking for our load information [hpc/info.c: info_recv_message()].

3.2.2 **Load Values**

Load values are collected on each node [hpc/balance.c: load_balance()].

1.local load - the actual load value for this node.

2.export load - the adjusted export load which is sent to other nodes

3.3.3 Load Vector Data

Each node keeps a fixed size (of length 8) vector containing

load information sent from other nodes. These values from the node's entire load vector are recorded [hpc/balance.c: load_balance()]. In addition for each node we record the node number , processor speed, export load, free slots - number of process "slots" available on the node.

3.3.4 UDP Message Data

In the openMosix extensions to the Linux kernel, UDP datagrams are used solely to send load messages

1.Size of most recent load message received [hpc/info.c: info_recv_message()].
2.Cumulative size of all load messages received. The size is constant at 40B for the standard kernel. [hpc/info.c: info_recv_message()].
3.Size of most recent UDP message sent [hpc/comm.c: comm_sendto()].
4.Cumulative size of all UDP messages [hpc/comm.c: comm_sendto()].

3.3.5 TCP Message Data

In the openMosix extensions to the kernel, all non-load messages are sent via TCP [hpc/comm.c: comm_send()].

1.Size of most recent openMosix TCP message sent.
2.Number of tcp messages sent.
3.Cumulative size of all tcp messages.

4 EXPERIMENTS

The issue of determining the "right" load vector size has received considerable attention. The work of Eager et al has shown that "simple adaptive load sharing policies, which collect very small amounts of system state information and which use this information in very simple ways, yield dramatic performance improvements." [7] Furthermore, they conclude, "simple policies offer the greatest promise in practice, because of their combination of nearly optimal performance and inherent stability." The load vector size and what it contains has changed many times throughout the history of MOSIX.

In addition to the length of the vector the decision as to what information to include in the exported load vector must be considered. The standard openMOSIX kernel only transmits information about the sending machine's load. An alternative approach is to propagate information about other nodes as well. We refer to this as a hearsay algorithm.[5] In this study the modified kernels utilize a hearsay scheme to propagate load information throughout the cluster. Each node keeps an array with *n* slots.. In each slot we keep the load information for nodes that map to that slot number modulo *n*. In addition to the load value, a time stamp is recorded. The time stamp indicates when the data was recorded by the node originating the information. All nodes record time stamp information. Nodes generate a time stamp when reporting about their own load. When a node receives information with a newer time stamp than the one currently held in the table at that slot it is replaced regardless of which node that it represents.[6] In the experiments we vary the value *n.* Thus, nodes not only inform two randomly selected nodes of *their* load but also propagate the latest hearsay information they have acquired about other nodes. The entire load vector is sent from each node to the other. Slot 0 is always reserved for a node to send its own load information. The remainder of the procedures remains as in the original kernel.

Since the various nodes are generating time stamp information it was necessary to decide whether to use logical or physical clocks. Given the availability of various commonly used time synchronization protocols we decided upon using physical clock data. The clock synchronization protocol we utilized was Network Time Protocol (NTP). The documentation from the NTP web site claims, "it provides accuracies typically within a millisecond on LANs." The ntpdate command was made to run on all nodes from cron once an hour. In addition immediately before each test was initiated the ntpdate command was executed again. We always used the ntpdate option to immediately change the time if the difference between a node and the time server is less than .5 second.

In the interest of space we will present only one battery of tests here.[7] The load injected into the system for this experiment on 20 nodes created 6 load spikes separated by 5 seconds each. Each spike starts 70 processes. The processes are all 100% CPU intensive. All processes from each spike live long enough for the system to have reached equilibrium from all spikes. Several tests were performed using modified kernels with varying vector sizes. For each modified kernel, the entire load vector is sent. Each test was run 5 times. The numerical results in the table below are an

5 Sometimes referred to as a gossip algorithm.

6 For example in our case with 20 nodes, if we use a load vector size of 8, 16 slots are shared by two nodes each and 4 slots are shared by three nodes each.

7 More test results and extensive multi-color graphs illustrating the results of each test can be found at *http://django.cs.wwu.edu/~mosixmods/60X/docs/*

average of the 5 test runs under that scenario. The load spikes occur on the same nodes for each test run.

- Test-A: load vector size = 8, send only own load.
- Test-B: load vector size = 20, send entire vector
- Test-C load vector size = 8, send entire vector
- Test-D load vector size = 12, send entire vector
- Test-E load vector size = 16, send entire vector

The test results given in the table below utilize the following terminology.

- **Recovery time**: The time it takes the cluster to return to a balanced state after a series of load spikes measured from start of first spike to last migration before the balance occurs.
- **Choose count per node**: The number of times the function *choose()* is run.
- **Migrations received per node**: The average over all nodes of the number of migrations received per node.
- **Ratio of choose count to migrations received**: Choose count divided by migrations received.

5 RESULTS

There were performance gains when using load vectors of size 8 and 12. Using vectors > 12 resulted in a performance decrease from the unmodified kernel. The recovery time figure in the table is the time at which the system recovered from the last spike. The spikes are arranged in the test script so that spike *n+1* occurs before any kernel variant is able to completely recover from the *n*th spike. As can be seen in the table, the recovery time using the standard kernel was 81.1 seconds. The version that sends a hearsay load vector of size 8 achieves a recovery time of 77.7, a 3.4 second improvement over the standard kernel. Increasing the load vector size to 12 and propagating hearsay information reduced the recovery time to 76.1 seconds, an improvement of only 1.6 seconds over the hearsay load vector of size 8.

When dealing with this type of system one must always be concerned to analyze the scalability of the methods employed. The most sensitive area is the scalability of the message traffic on the network as this is most often the resource most quickly saturated. openMOSIX utilizes UDP for all load information transfer. Since the load information is propagated once per second the number of messages in the standard kernel and the modified kernels tested remains constant. The size of the UDP load vector message for each test is given below. Each node actually sends two messages per second, since it sends to two other nodes, so the total UDP load vector traffic is twice the value shown in the table per second. Given that we were using Giga-bit links two messages of 300 bytes per second versus two messages of 40 bytes per second seems a reasonable expenditure for decreasing the time need by the system to respond to the spikes.

	Kernel	Load Vector Size	Recovery Time Seconds	Choose Count Per Node	Migrations Per Node	Choose Count / Migrations Received
A	2.1	8	81.1	74.5	13.4	5.6
C	5.0	8	77.7	66.7	13.7	4.9
D	6.0	12	76.1	73.1	13.7	5.3
E	7.0	16	87.4	85.9	14.7	5.8
B	4.3	20	96.0	88.6	14.9	5.9

Test	message size	Standard Kernel
A (unmodified algorithm)	40 Bytes	1
B (modified, vector size = 20)	732 Bytes	18.3
C (modified, vector size = 8)	300 Bytes	7.5
D (modified, vector size = 12)	444 Bytes	11.1
E (modified, vector size = 16)	588 Bytes	14.7

The kernels which performed worst called *choose()* more frequently, but did not result in the system as a whole migrating more processes. In our tests, no process used shared memory, had multiple threads or was manually locked as non-migratable. It is also possible that after a process is marked for migration it is never actually migrated because the receiving node can choose not to accept or because the local machine ended up with least opportunity

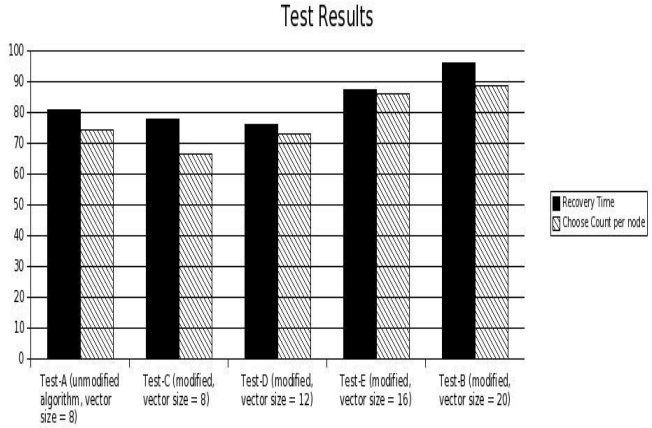
cost value. There could already be a process selected to be migrated and new load information triggered a *choose()* only to abort when the selected process' PCB is encountered.

One explanation of the results is that the kernels which send the largest load vectors trigger far more invocations of *choose()* because the extra information at hand makes it more likely that a remote node with a lower load value will be uncovered. Thus, they spend larger amounts of time deciding which process to migrate. The added information about the relative loads of the other machines enables us to make a better selection of the target node for migration. This is apparently outweighed at some point by the extra work involved in calling *choose()* more often. The kernels which send smaller vectors have "good enough" information to make a "good" decision. In other words, picking the best target processor is not necessary to have efficient recovery from load spikes.

The performance of the system improved when sending the entire load vector with hearsay information for load vector sizes of 8 and 12. It appears that there is a range in which additional information yielding better decision making concerning target node selection results in a performance increase in that the cluster is able to more rapidly diffuse spikes. It also appears that if we continue to increase the amount of information propagated the value of selecting possibly an even better target node is outweighed by the additional information triggering too many executions of process selection.

6 FUTURE STUDIES

Future kernel variants will investigate dynamically creating the load vector size and resizing if necessary during periods of low cluster load to fit the size of the cluster. The number of other nodes contacted will also be varied. Since it appears to be the excessive triggering of the *choose()* function when larger amounts of information are known about the system as a whole, this suggests that perhaps a modification to the choose algorithm and the way in which the load vector information is stored and manipulated may be worth investigating. We are currently implementing a statistical best fit algorithm. derived from the optimal stopping policy for Markov chains rather than traversing the entire PCB chain..



7 REFERENCES

[1] ANANE, R. AND ANTHONY, R.J. 2003. Implementation of a proactive load sharing scheme. Proceedings of the 2003 ACM symposium on Applied computing. Melbourne, Australia. 2003. 1038-1045.

[2] AMIR Y., AWERBUCH B., BARAK A., et al. 2000. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. IEEE Tran. Parallel and Distributed Systems 7, 760-768.

[3] BAR M. Introduction to openMosix. From the openMosix web site, in the "Documentation" section. ---23

[4] BARAK, A., GUDAY, S., WHEELER, R.G. The *MOSIX Distributed Operating System - Load Balancing for UNIX*. Lecture Notes in Computer Science Vol. 672 Springer 1993.

[5] Cheriton, D. R., "The V Distributed System," Communications of the ACM, Vol. 31, No. 3, March 1988, pp 314-333.

[6] Meehan, J. Michael, Philip A. Nelson, Elizabeth Lee Falta, A Retrospective of the DICE Distributed Operating System Project, *Conference proceedings of (CSITeA-03)*, *June 5-7, 2003.*

[7] EAGER, D.L., LAZOWSKA, E.D., AND ZAHORJAN, J. 1986. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. Software Eng. SE-12*. 662-675.

[8] GOSCINSKI, A., HOBBS, M., AND SILCOK, J. 2002. GENESIS: an efficient, transparent an easy to use cluster operating system. *Parall. Comput. 28*, 557-606.

[9] MALIK K, KHAN O, et al. 2005 Migratable sockets in cluster computing. *The Journal of Systems and Software 75*, 171-177.

[10] MAYA, ANU, ASMITA, SNEHAL, KRUSHNA. MigShm: Shared memory over openMosix. http://mcaserta.com/maask/Migshm_Report.pdf. April 2003.

[11] MORIN, C., GALLARD, P., et al. 2004. Towards an Efficient Single System Image Cluster Operating System. *Future Generation Computer Systems 20*, 505-521.

[12] MOSIX web site, http://www.mosix.org/

[13]openMosix web site, http://openmosix.sourceforge.net/

[14] Ousterhout, John, The Sprite Network Operating System, IEEE Computer, February 1988, pp 23-36

[15] POPEK, G. J., AND WALKER, B. J., The LOCUS Distributed System Architecture, Cambridge, Mass: The MIT Press, 1985