Improving Cluster Performancewith Dynamic Resource Management Algorithmsand a Generic Resource Management Framework

Joshua King

School of Computer Science and Software Engineering
School of Electrical and Electronic Engineering

Defence Science and Technology Organisation

Abstract

Distributed systems need to share tasks efficiently to achieve best performance. Process migration and dynamic resource allocation algorithms enable this. We investigated such algorithms, and created a generic resource management framework. We used openMosix to provide transparent process migration under Linux. Experiments that were run on 4-, 8- and 12-node clusters showed that performance depends heavily upon the process migration mechanism as well as the algorithms. Opportunity cost algorithms that monitored both CPU and memory consumption were the most efficient.

1.0 Introduction

Resource allocation is an important consideration in all computer systems design. Distributed computer systems, or clusters, are no different. Distributing workload between multiple systems is not trivial. We investigated techniques and algorithms for achieving balanced resource allocation in a cluster. We then developed an object-oriented framework to enable direct comparison of some of these algorithms. Lastly we ran experiments on a 12-node cluster to evaluate the algorithms and the framework. The eventual goal of this work is to develop a generic resource manager that can provide load balancing on a distributed computer architecture currently under development at the Defence Science and Technology Organisation.

2.0 Resource allocation and process migration

Resource allocation is an intractable problem, generally solved by either static or dynamic techniques. Static resource allocation involves taking a set of jobs and mapping them onto a fixed set of nodes ahead of time. Static techniques use *a priori* resource consumption information about jobs to determine a mapping. Dynamic techniques instead allocate resources on demand. Dynamic methods improve the load balance continuously by sharing statistical information. (Such methods may also be called adaptive.) Unlike static techniques, dynamic methods do not require prior knowledge of the tasks to be executed, nor the operating environment; a benefit that comes at the cost of overhead at runtime. Here we focus on dynamic resource allocation techniques.

Dynamic resource allocation requires functionality to move jobs between machines. Remote execution, often used by static resource allocation techniques, involves starting processes on a remote machine, however that process cannot be moved again. Remote execution, and therefore static allocation, may not be efficient with processes that vary their computational requirements during execution. Process migration adds the ability to move processes between machines during execution. This often involves copying some or all of the process' state between machines. Process migration can be repeated multiple times for the same process.

Such a distributed system offering process migration is the MOSIX for Linux (Barak & Braverman 1997) and openMosix (Bar 2005) extensions. These work in the operating system kernel and enable processes to migrate across a network. This involves splitting the process into two parts – a remote part, which contains all of the user-level code (such as calculations) and memory; and a local part, or deputy, which handles all of the kernel interaction (such as input and output). The deputy process enables the migration to be transparent to all processes. All of the kernel interaction occurs on the original machine enabling consistent access to resources such as files and sockets. OpenMosix was used for the experiment environment in this project.

3.0 Algorithms

The differentiator between dynamic resource management techniques is the algorithms that make decisions on when, what and where to migrate processes. We implemented a variety of algorithms with different properties from the literature and compared their behaviour under varying loads.

3.1 Distributed MINIX load balancer

Distributed MINIX (Tsai, Chiou & Jen 1994) is a variant of the MINIX operating system that offers both process migration and load balancing. We consider their load balancing algorithm. Their technique involves classifying systems and their processes. A system may have low (< 25% utilisation), normal or high (> 75% utilisation) load. A process may be classified as I/O bound, one which interacts with the kernel frequently, or CPU bound. It is most beneficial to reduce workload by migrating only CPU bound processes (Tsai, Chiou & Jen 1994). I/O bound processes may receive little speedup after migration because they continue to interact with (and get blocked by) the overloaded machine.

In Distributed MINIX both the sender and receiver can initiate process migration. A system with low load can request a process directly from a machine with a higher load (receiver-initiated), or a machine with high load can send a process to a machine that has advertised itself having a low load (sender-initiated). The load classification of each system is shared across the network.

3.2 Home model

Home model algorithms (Lavi & Barak 2001) directly consider transparent process migration in their calculations. The total workload when processing remotely is greater than when processing locally as long as local resources are available. The home model therefore leaves processes on their home system whenever possible. A penalty is added to a process' local cost when determining whether it should be migrated. Processes are still migrated if the home system is overloaded, but the algorithm selects those with the minimal penalty. Modelling of the additional latency caused by remote communications should make the algorithm better suited to process migration.

3.3 Hydrodynamic algorithm

The hydrodynamic algorithm (Hui & Chanson 1997) is a way of modelling workload as liquid flowing between different-sized cylinders. It visualises the workload in each system as the volume (or equivalently the vertical cross-sectional area) of some liquid filling a cylinder. The computer's relative performance, termed capacity, determines the diameter of the cylinder, enabling the algorithm's use in heterogeneous clusters. The goal is to equalise the heights of the liquid in all of the cylinders. The capacity and height of each node is shared with its neighbours.

3.4 Opportunity cost algorithm

The opportunity cost model (Amir et al. 2000) extends measurement to multiple resources. While the above algorithms only use a function of CPU load, opportunity cost adds memory consumption. This replaces ignoring or having independent algorithms for different resources. It calculates the sum of the proportion of consumption of each resource as each system's cost in the cluster. As the consumption of any resource increases, so does its cost. The algorithm aims to distribute processes such that the sum of these costs is minimised.

Here we consider the CPU and memory opportunity cost model. The cost of a system according to this model is given by the following formula:

$$c = n^{\frac{\text{used memory}}{\text{total memory}}} + n^{\frac{\text{CPU load}}{L}}$$

Figure 1: A cost function for the opportunity cost algorithm, *n* being the number of systems in the cluster and *L* being the maximum load.

This cost is summed over all systems. The system with the greatest cost attempts to reduce its cost by finding the best change in the overall cost from migrating a process. If such a reduction in the overall cost is possible, the migration is attempted.

4.0 Developing the framework

In addition to researching algorithms we needed an object-oriented framework for our resource manager. It needed to be flexible, allowing changes to be made, for example, to the migration algorithm. An extensible framework was one of the primary goals of this project, since the target environment for this research was not finalised.

Keeping the framework generic limited our access to information and services. To address this, we made abstract interfaces and corresponding concrete implementations for such services as statistics collection and process migration. OpenMosix and the Linux kernel provided these services in development and testing. We hope to extend this part of the framework to provide notification of job arrivals and departures, and to offer an application-specific migration service.

The framework is written to run in a single thread to make it more robust. There is a timer that triggers each object to update its state at regular pre-determined intervals. This simulates concurrent execution without having to program against race conditions.

The networking infrastructure uses a pair of classes, a server and a client. We used simple group communication implemented over multicast UDP sockets to share statistics. This works seamlessly while the cluster is self-contained on a single network segment, as was the case for our experiments. Being multicast we don't need to know all of the hosts to send to, however we can still determine the number of them and their addresses by monitoring the source addresses of statistics packets. We combat the non-guaranteed delivery of UDP packets by containing statistics within single packets.

This class diagram shows the framework we developed graphically. Under each abstract (*italicised*) class are concrete implementations providing that service that are not shown.

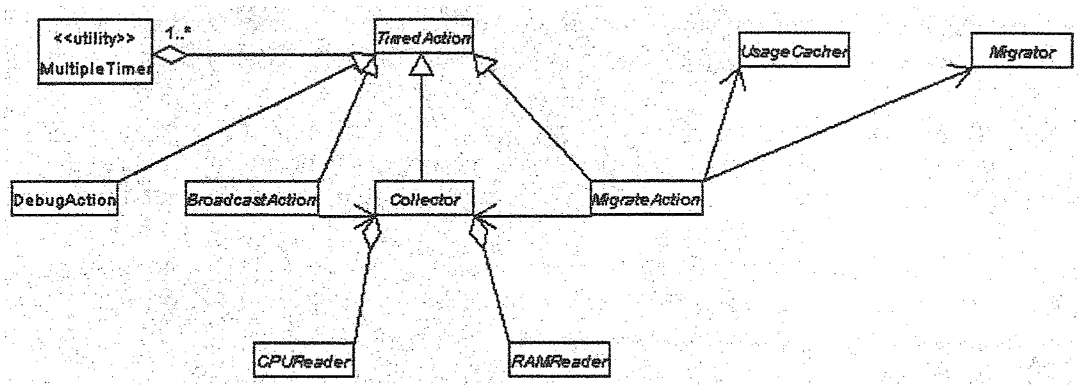

Figure 2: The structure of the abstract framework developed, operations and attributes have been hidden from the diagram.

The framework was realised in the C++ programming language. We made extensive use of the Standard Template Library (STL) for its data structures and their automatic memory management. At this stage only Linux and openMosix statistics collection and migration have been implemented.

5.0 Experimentation and results

Lastly we needed to compare the algorithms that we had implemented. We derived effectiveness from the execution time and amount of work done, with the hope of ranking the algorithms. We also tested the scalability of the framework and algorithms across clusters of three sizes.

5.1 Test environment

We set up a cluster of 12 dual-processor Pentium III servers with 1GB of RAM each. We used a modified ClusterKnoppix (Vandersmissen 2004) environment. ClusterKnoppix is a Linux live CD providing openMosix that runs entirely from memory. OpenMosix's automatic load balancing was disabled during all of the experiments. An additional machine with openMosix disabled recorded the statistics and debugging output from each machine. We monitored the load during and execution time of the test along with the resource consumption of the resource managers. To test scalability we compared 4- and 8-computer subsets of the cluster, modifying the openMosix configuration accordingly.

We ran benchmarks from the openMosix stress test suite (Rechenburg 2005). The stress test suite was altered to allow each test to be executed independently. We ran the following subset of tests:

- **Distkeygen**: Generates a large number of RSA encryption keys in parallel processes.
- **Forkit**: Quickly creates many processes that perform computations but not I/O.
- **Portfolio**: A stock market simulation written in the Perl interpreted language that splits itself into multiple processes.
- **Timewaster**: A test which runs frequently calls the clock in multiple parallel processes.

Each test adjusts its workload based on the number of processors in the cluster. Therefore, in the 12-system tests more work is required than the comparable 4-system test, so absolute execution times and workloads are not directly comparable.

5.2 Test results

The method we used to compare results involved determining the execution time and calculating a *workload ratio* for each test. We derived this equation for this purpose:

$$\textit{workload ratio} = \frac{\sum_{i=1}^{N}\sum_{t=1}^{T} CPU\%(i,t)}{100T}$$

Figure 3: Formula for calculating workload ratio, where N is the number of systems and T is the execution time in seconds.

The workload ratios for each of the 12-system tests are graphed below. Note that the baseline values are the result of running the benchmark with no process migration occuring at all (as if running on a single machine). Some of the forkit tests did not complete. We also tested two variants of the opportunity cost algorithm. For workload ratio, a larger number indicates higher efficiency and shorter execution time.

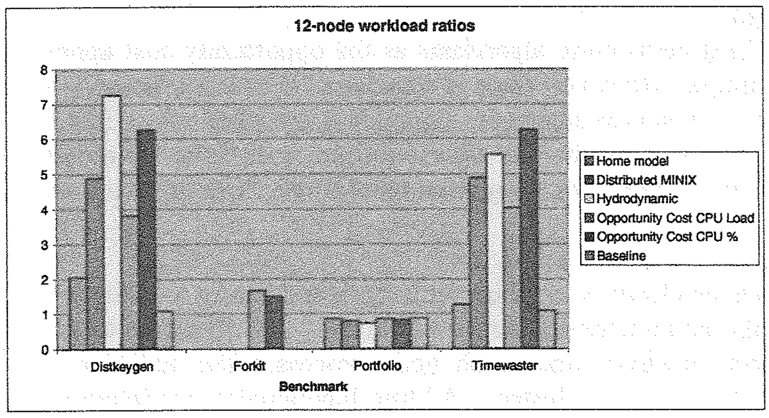

Figure 4: Graph of workload ratios derived from 12-node experiments

5.3 Analysis

All algorithms migrated processes, increasing the efficiency of the cluster for most tests. The worst performing algorithm was the home model. We believe this to be the result of our implementation rather than the theory. Having Linux initiate processes, forcing the home system before we can decide to move them, reduces performance in all cases. A home model implementation especially should perform better if triggered upon process creation and termination, but in our framework this is left for future work.

The two opportunity cost variants show the difference between two methods of determining CPU consumption, the methods used being the one-minute UNIX load average and the ratio of idle CPU cycles to total CPU cycles over a second (CPU %). The first can be read directly from the kernel, whereas the second requires some calculations. The more parallel benchmarks, distkeygen and timewaster, show the CPU % method outperforming the load average method, verifying previous results (Kunz, 1991).

All of the algorithms handled the portfolio benchmark very poorly. All execution times were longer than the baseline, and the workload ratios were rarely better. The low baseline workload shows that a single node was not taxed by this test. This suggests the algorithms may be too aggressive at balancing load when it is not necessary.

The forkit test was the only benchmark where most algorithms failed. We discovered the cause to be the home node's RAM becoming full. Since the test systems have no virtual memory, exhausting the memory caused the tests to be killed. Some algorithms managed to migrate some processes away before the memory was filled and thus completed the benchmark. All of the algorithms completed on the 4-system cluster, due to the lower total workload, however both the home model implementation and the baseline test failed to complete on the 8-system cluster. Likewise, only the opportunity cost variants completed on the 12-system cluster, which can be due to the inclusion of memory consumption in their calculations.

We also tested for scalability by analysing the CPU and RAM consumption of the resource managers during the tests. As the number of systems increased, the CPU usage increased, suggesting an upper limit on the number of systems that can be managed in this fully-connected approach. For 12 systems, the average CPU consumption during the tests rarely exceeded 4%.

6.0 Conclusions

The results show the best-performing algorithms as the opportunity cost approach, which balances both CPU and RAM usage. However, there is a trade-off between efficiency of the cluster and the CPU consumption of the resource manager. Simpler approaches may often be more appropriate. We also noted with closer analysis that the algorithms are sensitive to the ordering of processes, and are heavily dependent upon the process migration mechanism, both of which were strictly controlled in these experiments.

The framework that we developed will be useful in both continuing to analyse better models and for implementation into the architecture under development. The framework is currently suitable for moderate-sized clusters, however since each node processes the statistics of every other node, changes would be needed for large clusters. Adding functionality for intercepting process creation and termination events would enable an improved implementation of the home model algorithms.

7.0 References

Amir, Y., Averbuch, B., Barak, A., Borgstrom, R.S., & Keren, A. 2000 'An opportunity cost approach for job assignment in a scalable computing cluster,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no.7, pp. 760-8.

Bar, M. (18 August 2005) *OpenMosix, an open source Linux cluster project*, [Online]. Available from: <http://openmosix.sf.net> [29 August 2005].

Barak, A., La'adan, O. & Shiloh, A. 1999, 'Scalable cluster computing with MOSIX for Linux,' in *Proceedings of Linux Expo '99*, Raleigh, USA, pp. 95-100.

Hui, C. & Chanson, S. 1999, 'Hydrodynamic load balancing,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 11, pp. 1118-37.

Kunz, T. 1991, 'The influence of different workload descriptions on a heuristic load balancing scheme,' *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725-30.

Lavi, R. & Barak, A. 2001, 'The home model and competitive algorithms for load balancing in a computing cluster,' in *21st International Conference on Distributed Computing Systems*, Mesa, USA, pp. 127-34.

Rechenburg, M. *The openMosix stress-test*, Version 0.1-4.1 [Online]. Available from: <http://www.openmosixview.com/omtest/> [29 August 2005].

Tsai, S.R., Chiou, J.-T., & Jen, H.-T. 1994, 'Load balance facility in distributed MINIX system,' in *Proceedings of the 20th EUROMICRO Conference. System Architecture and Integration*, Liverpool, UK, pp. 162-9.

Vandersmissen, W. (20 June 2005) *ClusterKnoppix – Main*, Version 3.6-2004-08-16-EN-cl1 [Online]. Available from: <http://bofh.be/clusterknoppix/> [29 August 2005].