Developing and Analysing Dynamic Resource Management Algorithms with a Generic Framework

Joshua King

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2005
Abstract

Distributed systems need to share tasks efficiently to exploit their performance. A combination of process migration and dynamic resource management algorithms can enable this. We investigate one such migration technique and five algorithms in order to create a generic resource management framework for distributed systems. We use openMosix’s transparent process migration under Linux for experimentation.

Experiments that were run on 4-, 8- and 12-node clusters showed that performance depends heavily upon process migration, and less so on the algorithms. Some algorithms also monitor multiple resources to improve performance. Scalability of the framework is an important consideration, and we found that even with distributed control a completely connected network would soon become unmanageable as the number of nodes is increased. We find that the opportunity cost algorithms are the most consistent but upon consideration of scalability suggest that simpler algorithms, such as the distributed MINIX load balancer, are desirable.

Keywords: load balancing, dynamic resource management, algorithms, user-space, MOSIX

CR Categories: C.2.4 Distributed Systems, C.4 PERFORMANCE OF SYSTEMS
Acknowledgements

This project was conducted under the CEED project grants scheme as a studentship between the student, the CEED Office at The University of Western Australia and the Maritime Operations Division (MOD) of the Defence Science and Technology Organisation (DSTO) at HMAS Stirling, Western Australia.
Contents

Abstract ii
Acknowledgements iii

1 Introduction 1

2 Literature Review 3
  2.1 Process migration 3
  2.2 Algorithms 4
    2.2.1 Hydrodynamic algorithm 6
    2.2.2 Opportunity cost 7
    2.2.3 Home model 8
    2.2.4 Distributed MINIX 9
    2.2.5 Memory ushering 9

3 Implementing the Framework 11
  3.1 Design 11
  3.2 Implementation 15
  3.3 Algorithms 16
    3.3.1 Hydrodynamic algorithm 16
    3.3.2 Opportunity cost algorithm 17
    3.3.3 Home model 18
    3.3.4 Distributed MINIX load balancer 19
    3.3.5 Memory ushering 20

4 Experiments and Data 21
  4.1 Environment 21
  4.2 Results 23
5 Analysis

5.1 Comparing the algorithms
   5.1.1 Algorithm observations
   5.1.2 Benchmark observations

5.2 Scalability

5.3 Further observations

6 Conclusions

A Original Honours Proposal

A.1 Background

A.2 Aim

A.3 Method

A.4 Software and Hardware Requirements

B Test Environment

B.1 Modifications to ClusterKnoppix

B.2 Modifications to openMosix test suite
List of Tables

2.1 Algorithm breakdown . . . . . . . . . . . . . . . . . . . . . . . . . 6
4.1 CPU consumption . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overall Activity Diagram</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Framework Class Diagram</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>4-node workload ratios</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>4-node execution times</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>8-node workload ratios</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>8-node execution times</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>12-node workload ratios</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>12-node execution times</td>
<td>27</td>
</tr>
<tr>
<td>4.7</td>
<td>CPU Utilisation Comparison</td>
<td>28</td>
</tr>
<tr>
<td>4.8</td>
<td>CPU Consumption by Algorithm and Number of Nodes</td>
<td>29</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Resource allocation is an important consideration in all computer systems. Distributed computer systems likewise have this need, but it is less trivial to distribute workload across multiple machines. In this project we investigate some current algorithms and techniques for dynamically achieving balanced resource consumption in a computing cluster. We then develop an object-oriented framework to enable direct comparison of some of these algorithms.

Research into improving resource allocation is ongoing since the optimal algorithm cannot be achieved. The ideal algorithm would have knowledge of tasks' future behaviours, and also consume none of the resources we are attempting to share. While this is unachievable, as computational power increases we can afford to devote more processor time to scheduling in the hope that more sophisticated algorithms will better balance more resources in the distributed system.

Our primary goal is to develop a resource manager framework that is suitable for integration into the Defence Science and Technology Organisation’s (DSTO) Generic Open Architectures for New Naval Applications (GOANNA) project. The GOANNA project is an open standards based computing architecture being designed to support component based combat system applications running on heterogeneous clusters of computers. These components can be developed independently and later integrated into a working system. Combat systems applications involve continuous execution of a variety of tasks, such as tracking and sonar, which are resource-intensive and are therefore suited to distributed or parallel processing.

We intend to determine the effectiveness of some existing load balancing policies and algorithms, while developing a flexible framework that can be implemented for the GOANNA open architecture. This framework will be modular to allow for differences between our testing environment and the final implementation environment to be defined by the GOANNA project. The GOANNA project also intends to provide its own migration technique, so for these experiments we focus on analysing the algorithms rather than the migration technique in use.
In this paper we first review the literature in this area (Chapter 2). Here we look at the methods for dynamic resource management, the way computer systems support it, and some algorithms from the literature. In Chapter 3 we explain the design chosen for our framework, how it was developed, and how we translated the algorithms in Chapter 2 into the working implementations used for comparison. Chapter 4 describes the experimentation environment and the results of this testing. These results are then analysed (Chapter 5), in terms of both efficiency and scalability. Lastly we make some conclusions and discuss some opportunities for future work (Chapter 6).
CHAPTER 2

Literature Review

Resource allocation techniques can generally be split into static and dynamic techniques. Static resource allocation involves taking a set of jobs and mapping them onto a fixed set of nodes. These techniques use *a priori* information about the resource consumption of these processes in order to determine the most favourable mapping. Dynamic resource allocation allocates resources at run time. Dynamic methods can continuously work to improve the balance of resource allocation across the networked computers by sharing statistical information. (Such methods may also be called *adaptive.*) Dynamic methods do not require knowledge of the tasks to be executed, nor the operating environment; a benefit coming at the cost of overhead at runtime. Here we focus on the use of dynamic resource allocation techniques.

2.1 Process migration

In order to fully support dynamic resource allocation, there must be functionality enabling jobs to move between machines. Remote execution is such a technique, however it can only be applied when a job is first initiated. Remote execution may not lead to the best resource allocation, especially when long-running processes, or processes with varying resource consumptions are involved. Many distributed systems now implement process migration in addition to remote execution. Process migration is a technique offered by distributed operating systems that enables processes to be moved between machines transparently during their execution. This involves copying the process’ state to the new machine.

One such distributed system, built around process migration, is the MOSIX family of operating system extensions. Barak et al. [4] have developed MOSIX for the Linux operating system, in addition to many other UNIX-based environments. MOSIX, and the open-source openMosix variant [2] are extensions to the kernel that enable processes to migrate across a network. This approach involves splitting the process into two parts – a remote part which contains all of the
user-level code and memory and a local part, termed the deputy, which handles all of the kernel interaction. The deputy process is what enables the migration to be transparent to all processes. All of the kernel interaction is forwarded to the deputy which runs on the Unique Home Node (UHN) of the process. This ensures consistent access to resources such as files and sockets.

Remote Unix [12] is a similar system for achieving process migration. Here the processes are compiled against a library that provides indirect access to the system calls. When the process is migrated to run remotely, the library uses Remote Procedure Calls across the network to access the information from the node on which the process was initially executed. As in MOSIX, there is a shadow process which remains on the original machine to service these remote procedure calls.

### 2.2 Algorithms

The performance of any distributed system is constrained by the efficiency of its resource management. Therefore improving these algorithms is an active research area. There are many different techniques for achieving balanced load across multiple machines, here we discuss five of them.

Firstly, there are a few different characteristics of dynamic resource management algorithms. Shivaratri et al. [14] identify many properties of load balancing algorithms, and we note some more from the comparison of algorithms herein. These characteristics include:

- Centralised versus distributed
- Sender-initiated versus receiver-initiated
- Job-centric versus load-centric
- Multiple resources managed by a single algorithm versus multiple algorithms

Centralised algorithms are those that send all resource usage information to a single machine that is solely responsible for making decisions about resource allocation and process migration. These algorithms are vulnerable by having a single point of failure. On the other hand, distributed algorithms share the decision-making between all nodes in the cluster. Distributed algorithms can be more efficient even if they do not use information from all of the machines. These algorithms can dynamically improve their decisions, continuously finding
new local minima of resource consumption as workload propagates across the cluster.

Algorithms can be sender-initiated, receiver-initiated or both. This refers to the direction in which migrations are triggered. The sender-initiated model involves an overloaded system trying to offload processes to less loaded systems. Conversely, the receiver-initiated model involves the less-loaded machines seeking load from the heavily-loaded machines in the cluster.

Job- and load-centric algorithms differ in terms of the actions that trigger them. Job-centric algorithms focus on job arrivals and departures. Load-centric algorithms are instead triggered by the results of some formulae that calculate the loads on a machine, compared against some thresholds. The theoretical basis of most algorithms follow the job-centric model.

Job-centric algorithms can also be classified according to whether they cater for temporary or permanent jobs. Temporary jobs are those which have a finite amount of work to do and will then terminate. Permanent jobs are those not expected to terminate under normal circumstances. Some simple job-centric algorithms may work under the assumption of permanency, that is not considering job departures. This reduces the amount of information that must be managed. It is a valid simplification because the amount of work remaining for a process often cannot be determined, and because some processes never terminate.

Load-centric algorithms can also be classified by how multiple resources are handled. A resource is any finite quantity that must be shared between jobs on the cluster. CPU load and main memory usage are the most commonly modelled resources, however as the algorithms increase in sophistication, some are introducing the measurement of network and disk I/O bandwidth in use by the job. Note that some of these additional measurements need to be handled carefully when used in combination with process migration, since a disk access by a migrated process incurs a network cost for the request and receipt of the data when the process is not running in the node hosting the data. As such, these calculations become increasingly complex. The alternatives are to ignore additional resources, or to use separate algorithms for them. Simpler load balancing strategies tend to ignore resources other than CPU load.

Lastly, algorithms for dynamic resource allocation can be described by their information, transfer, selection and location policies [14]:

• **Information policy**: Defines what facts (such as load statistics) are shared between the machines, and what information needs to be stored.

• **Transfer policy**: Determines when jobs may be migrated (in what state the machine must be in).
• **Selection policy**: Determines how jobs to be transferred are chosen.

• **Location policy**: Determines which machine will receive the job.

We have selected five dynamic resource allocation algorithms for comparison. These algorithms can all be implemented in a distributed fashion. All of the algorithms except memory ushering measure CPU usage. They differ in the metrics they use to determine workload, the way they select nodes to transfer load to, and who is responsible for the load transfers. Memory ushering is instead an algorithm specifically for balancing free memory. Table 2.1 indicates the basic characteristics of each algorithm.

The algorithms are the hydrodynamic model, the opportunity cost model, the home model, the distributed MINIX load balancer and the memory ushering algorithm. Of these, the first four are intended as standalone policies for load balancing. The opportunity cost model monitors both CPU usage and memory consumption. Memory ushering measures only memory consumption and is intended to be used alongside an existing load balancing strategy.

<table>
<thead>
<tr>
<th></th>
<th>Hydrodynamic</th>
<th>Opportunity</th>
<th>Home</th>
<th>Distributed</th>
<th>MINIX</th>
<th>Memory Ushering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralised</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sender-initiated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Receiver-initiated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job-centric</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Load-centric</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple resources</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Breakdown of algorithms into categories

### 2.2.1 Hydrodynamic algorithm

Hui and Chanson’s hydrodynamic algorithm [7] is a way of modelling workload as liquid that can flow between different-sized containers. It visualises the workload in each node of the cluster as the volume (or equivalently the vertical cross-sectional area) of this liquid filling a cylinder. The hydrodynamic strategy caters for heterogeneous clusters – those having computers of differing performance (varying in the number of processors or clock speed for example). The node’s relative performance, or capacity, determines the diameter of the cylinder. The
goal of the algorithm is to equalise the heights of the liquid in all of the cylinders that represent the cluster.

It is a load-centric algorithm that can be implemented in a centralised or distributed manner. As with all distributed algorithms, some information must be shared. For the hydrodynamic algorithm, this is the capacity and current height of each neighbour node.

Each node attempts to reduce its own workload by sharing it with its neighbours. The algorithm flows the liquid from the current node into a container holding the node’s and its neighbours’ cylinders in ascending order of current heights. The final height of the liquid poured determines how much workload to transfer and to which nodes. That is, the fair amount of workload is determined and at least this much remains at the node. Any remaining workload is distributed to neighbours in order of increasing current workload such that workload is not sent to nodes where it would exceed the determined fair level.

In real-world situations, workload cannot be infinitely divided. To account for this case, the algorithm is refined such that the amount of liquid to be transferred accumulates until it exceeds a threshold value. Once a job is transferred, this counter is decremented and required to accumulate again before another transfer to that neighbour will occur.

The work on the hydrodynamic model does not restrict what quantity should be used as the amount of workload on a node, but we assume some function of CPU load.

The hydrodynamic algorithm has also been extended to work across wide area networks, such as the Internet [6].

2.2.2 Opportunity cost

Amir et al.’s opportunity cost model focuses on extending the measurement of workload to more than one resource. Commonly algorithms use a function of only CPU load or availability for their workload calculation. Another technique is to have multiple prioritised algorithms for different resources. Some versions of MOSIX used a pair of algorithms [3], implementing a CPU load balancing algorithm and the memory ushering algorithm. The opportunity cost approach considers instead the proportion of consumption of each resource. It combines these ratios to derive the cost of each node in the cluster. As the workload on a node with respect to any of the monitored resources increases, that node’s cost increases. The algorithm aims to distribute processes such that the sum of these costs is minimised.
Amir et al.’s implementation of opportunity cost considered CPU load and memory consumption to replace the two separate algorithms in MOSIX. Further extensions have since been made to the approach, including the introduction of disk and network I/O to the calculation by Keren and Barak [8]. Here we consider the simpler CPU and memory only model.

Node $i$’s cost is given by the following formula, where $n$ is the number of nodes known to the node and $L$ is the maximum load:

$$cost_i = n \frac{\text{used memory}_i}{\text{total memory}_i} + n \frac{\text{CPU load}_i}{L_i}$$ (2.1)

This cost is summed for all of the nodes. The node that has the greatest cost will attempt to reduce the overall cost by determining the change from moving each job to a different node. If it finds a job for which the cost improves, that job will be migrated.

2.2.3 Home model

Lavi and Barak [11] introduced the home model algorithm to improve decision-making when transparent process migration is being used. Transparent process migration, where the process is unaware of any changes to its operating environment, is generally achieved by splitting the process into two parts. The user computation part, is sent to a remote machine for execution. A shadow process is left at the originating machine. Most system calls are forwarded back into the shadow process for execution. There is a trade-off in this technique, and it is this that motivates the home model algorithms. There will be greater total workload involved in processing remotely rather than locally, if the resources are available locally. For this reason, the home model assigns processes to their home node as often as possible. This is achieved by assigning some penalty (an additive or multiplicative factor) to the job’s current local cost when determining whether it should be migrated. Processes are still migrated should the home node become overloaded, but the algorithm is careful to select those with the minimal penalty for migration.

The home model suits a process migration framework better than other algorithms because it models the additional latency that occurs with communications between a remote process and its deputy on the home node. Another approach that considers a penalty of process migration is the Distributed MINIX load balancer.
2.2.4 Distributed MINIX

Tsai et al. [15] developed a version of the MINIX operating system which was able to handle process migration and load balancing. The load balancing algorithm embedded into their version of the MINIX operating system enabled automatic process migration to occur. Their technique involves classifying nodes and their processes into a small number of categories. A node may have low (< 25% utilisation), normal or high (> 75% utilisation) load. A process may be I/O bound or CPU bound. Tsai et al. define an I/O bound process as one which interacts with the kernel (via system calls) for more than $\frac{1}{11}$ of its execution time. This is measured using the ratio of user time to system time. Tsai et al. show experimentally that it is most beneficial to reduce workload by migrating only CPU bound processes. I/O bound processes receive little speedup after migration because they continue to interact with (and potentially get blocked by) the overloaded machine.

Distributed MINIX also allows processes to migrate automatically in two ways. Firstly a system with low load can offer directly to share the workload of a machine with a higher load (receiver-initiated) and secondly the machine with the high load can send a process to a machine that recently advertised itself having a low load (sender-initiated).

The algorithm distributes only the load classification of each node. This determines other neighbour machines’ behaviours. To reduce overhead the classification is only sent when it is low or has changed. Repeatedly sending the low load message doubles as an invitation for more workload to be sent to that machine.

2.2.5 Memory ushering

Barak and Braverman [3] developed the idea of memory ushering. This is designed as a secondary algorithm for the management of memory consumption in the cluster. It is intended to operate alongside some other load balancing algorithm for managing CPU usage and is only triggered if the amount of free physical memory decreases below a predefined threshold.

The reasoning behind implementing a memory ushering algorithm is to maintain the performance of the cluster as much as possible without having machines resorting to using virtual memory. Virtual memory is many times slower than physical memory.

This algorithm was designed with MOSIX in mind. As such, it is distributed, load-centric and sender-initiated – machines with the least free memory attempt
to send jobs to machines with more available memory.

The amount of free memory in the nodes is broadcast and stored, allowing decisions to be made locally when a node’s memory becomes almost full. The algorithm tries to reduce the number of migrations it would take to reduce memory consumption by transferring the largest process it can. It chooses the largest process to fit in any other node and then finds the node that process fits best. In case of a tie, the node with the least load is chosen.
CHAPTER 3

Implementing the Framework

For the purposes of getting a working product and testing the algorithms, we spent some time developing an object-oriented framework for our dynamic resource manager.

3.1 Design

The framework we have designed allows for the implementation of variants of all of the above algorithms and integration with different process migration strategies. The framework is designed to be generic in that it allows for the replacement of each part with a different variant. For example, the statistics collector hierarchy can be replaced by one which collects data from a BSD operating system rather than a Linux operating system, and the migrator can be substituted with an interface to a different migration system. However at this stage only Linux and openMosix statistics collection and migration components have been implemented for this framework.

The final resource manager for the GOANNA architecture requires this modularity since a goal of the GOANNA project is to enable interoperability in heterogeneous clusters. This modularity at compile-time allows this to occur since configuration of the resource manager would cause only the appropriate subclasses to be built on a given system. Having abstract interfaces for each of these classes protects the rest of the framework from needing to know which particular subclasses have been implemented.

By building the framework outside of the operating system kernel, as well as attempting to keep it generic, we were limited by what information and services we have access to. For example, process migration is external to the framework. This may be either built into the operating system kernel, or as an application-level service. We relied upon openMosix and the load information provided by the Linux operating system in development and testing, but left the option for
implementing other interfaces. The GOANNA project intends to provide its own application-level service for enabling component migration, however an implementation was not available during this project. Better tracking of job arrivals and departures is also intended to be provided by this interface. An interface to this service will be implemented in the future.

The framework is also designed to be run as a single thread to make it more robust. There is a timer system that triggers each object to update its state on an interval defined by each object. Every object that needs to be triggered inherits from the TimedAction class and is sent to the MultipleTimer class at startup. The MultipleTimer class acts as a singleton, that is there may only be a single instance of it in the process. The TimedAction interface provides a method which is used as an entry point. When each object’s interval expires, the MultipleTimer class calls this method on each object in turn, relinquishing control of the resource manager to the TimedAction instance. Thus, TimedActions should be designed to execute as quickly as possible. This implements simple co-operative multitasking within the resource manager.

The dynamic model of the system, showing a normal set of tasks being run from the timer system, is given in Figure 3. These tasks can be varied. For example, during testing of the framework, additional debugging actions were inserted into the system. Likewise multiple algorithms could be executed, as is necessary to use the memory ushering algorithm.

The collection of statistics is the responsibility of the Readers. Currently there are readers for CPU and RAM statistics. These Readers provide per-process and overall statistics where possible. The storage of these local statistics is handled by a Collector which aggregates the data for usage by the algorithms and the network. The Collector provides the detailed set of local statistics, but also maintains a summary set of statistics. The summary set is intended to be only those statistics that need to be distributed across the network.

An example of abstraction in the framework is the reading of CPU statistics. It is abstracted into a CPUReader class, with the concrete LinuxCPUReader subclass providing the functionality required under Linux. The OpenMosixCPUReader class further extends this functionality when compiled for an openMosix-enabled kernel. A similar structure is applied to collecting RAM statistics.

The networking infrastructure is designed to use a pair of classes, one for sending statistics and one for receiving them. Classes that inherit from BroadcastAction implement the broadcast of statistics, and those that inherit from UsageCacher receive them. It is important that these be paired up correctly because together they (not the framework itself) define the communications protocol in use.
Figure 3.1: The model for the dynamic activity of the resource manager framework. The activities can be varied.
The currently implemented technique for broadcasting statistics is a simplified form of group communication implemented over multicast UDP sockets. The cluster was self-contained on a single network segment for our experiments. Multicast UDP enables easy management of the distribution of statistics. We tell the kernel to only send us packets received on the UDP socket that weren’t sent by us, eliminating receipt of our own statistics, and the nature of multicast means we don’t need to know all of the hosts to send to. We can determine which hosts are in the cluster by logging the source addresses on the statistics packets sent to the multicast address. We determine the number of hosts by the number of unique source addresses. Both of these are required since the GOANNA architecture intends to support dynamically changing clusters, hosts may come up and down during the life of resource manager. The unreliable nature of UDP is addressed by requiring that statistics messages be self-contained in a single UDP packet. GOANNA has a group communications standard (based upon the Spread toolkit \[5\]) that will be implemented in the future.

Lastly are the MigrateActions and the Migrators. The MigrateAction subclasses are the algorithms that determine when, what and where to migrate. The Migrators are responsible for how the migration is achieved. By providing an abstract interface to the process migration system we are able to separate the decision-making from the underlying architecture, allowing that to be changed as necessary.

The class diagram (Figure 3.2) below describes the above relationships graphically. For clarity and space all operations and attributes have been removed from the diagram. Only the abstract classes (the interfaces) are shown here.

Figure 3.2: This is a UML class diagram showing the abstract class hierarchy developed towards implementing a dynamic resource manager.

The development of an extensible framework was one of the primary goals of this project, as the GOANNA target environment of this research is neither
finalised nor in use. As the GOANNA project develops, the framework will be modified to collect statistics directly from the GOANNA components, use a standard technique for group communication and implement its component migration technique. These interfaces will supersede the Linux and openMosix interfaces used for this project. The framework may also be extended to manage additional resources made available in these interfaces.

3.2 Implementation

The framework was realised in the C++ programming language. We made extensive use of the Standard Template Library (STL) for its data structures and its provision for safer memory management. Memory management is of importance to all applications, especially those expected to run continuously in the background. The STL provides easy allocation and de-allocation of variable length lists and maps (among other data structures) that reduce the chance of memory problems in the application.

As for process migration, we limited ourselves to the freely-available openMosix [2] kernel modification to Linux. We did this to reduce the number of variables in our experimentation, while noting that there are different techniques available for process migration. openMosix was chosen because it allowed us to disable its own built-in process migration and perform our own migration via Linux’s /proc file system. openMosix can transfer most Linux processes between hosts of the same architecture and operating system. It cannot transfer all processes – those which have a tight dependency upon the kernel may not migrate. openMosix also allows a process to disable its own migration. This feature is exploited by the resource manager, since it should always remain on the original machine. Our framework performs some basic checks as to whether migration will be successful, however in general the algorithms assume that every process in the process list for a node is a candidate for migration. DSTO plans to implement its own migration system as part of its developing GOANNA architecture.

Because of the software engineering focus of this project, much emphasis was placed upon the object-oriented design, testing, documentation, and management of rationale during this project. The emphasis on design was a major part of the work as described above.

DSTO has a specialised system for documenting project decision-making and sharing documents called the Job Management System (JMS). This was used extensively throughout the project to document the project and store notes about decisions. Additionally it was used to retrieve other information from
past projects to prevent making the same decisions twice. It can also be used for team management, however this was not exploited in this project as the task was limited to myself and my mentor.

In terms of testing, a heavy emphasis was made on unit testing in the early stages of development, with each class being tested independently. Because of the distributed systems focus and need for multiple systems to fully test functionality, emulation provided by VMWare Workstation \[17\] was used. Later this expanded into integration testing with different combinations of components being deployed. Additional debugging components were built and integrated to analyse problems when they occurred. These components can now be enabled or disabled as necessary to allow continuous testing as changes are made.

### 3.3 Algorithms

We have implemented the five algorithms discussed in the literature review (Chapter 2, above). Their policies are mostly similar to those in the referenced papers though some changes have been made to accommodate our framework. Their policies are described below.

The framework cannot yet give us job control to the level of controlling which node processes are initially executed on. This is due to the implementation being a user space application, and process initiation not being exposed to us by the kernel. As such, the transfer policies of these implementations are the same. For all new processes, the node is a receiver, that is, they will start to be executed there and be subject to selection for process migration the next time the algorithm is executed.

#### 3.3.1 Hydrodynamic algorithm

Our hydrodynamic implementation follows Hui and Chanson’s \[7\] HTERO-LB implementation. On the basis of their experimentation, and that we have a complete (fully connected) network, we use a constant threshold value for job transfer of 0.4.

- **Information policy:** The hydrodynamic algorithm uses the speed of the nodes, which we distribute across the network, to determine the capacities of the nodes. We use the CPU load average as the workload. The heights are the ratio of these two quantities. Counters of load outstanding on each
edge in the network are also shared when they change (over an independent unicast UDP channel to the particular host) due to process migration.

- **Selection policy:** The process chosen for transfer to the remote node is not defined by this algorithm, so an arbitrary process is chosen. In our implementation, we choose the first process in decreasing numerical order, likely the most recent to be started or migrated to this host, that migrates away successfully.

- **Location policy:** The possible locations for migration are those whose counter value exceeds the threshold value. Each of these nodes get one process and the counters are subsequently reduced by 1. Counters can become negative in this algorithm.

### 3.3.2 Opportunity cost algorithm

The opportunity cost implementation we used measured CPU usage and memory consumption. We modified Amir et al.’s formula for the cost of a particular node. The CPU load portion of the calculation was implemented in two ways. First we implemented it using the percentage of CPU utilisation, then we tried to extend it to the CPU load average but had difficulty with the $L$ quantity described in that paper. The behaviour of the algorithm when using a locally-calculated maximum load for $L$ was such that process migration appeared to be inhibited completely in early testing, so instead we took the maximum load seen globally across all of the nodes participating in the cluster. This simplifying assumption is valid for a homogeneous cluster (that is, one where all of the nodes offer the same performance) such as ours, but may not be suitable in an arbitrary heterogeneous cluster.

Another concern for the use of the CPU load average is Kunz’s experiments on CPU usage metrics. Kunz discovered that the one-minute load average reported by UNIX operating systems was the least efficient indicator of workload mainly because it reacts to new load too slowly. To investigate this result we maintained two variants of the opportunity cost algorithm that differed only in the method of capturing CPU consumption.

Because of the potential for working in a heterogeneous environment, we made sure that the marginal cost calculation which determines the selection and location of migration accounted for variations in total memory and CPU speed. The percentage of memory in use at the current node was converted into the percentage of memory at the new node. The corresponding adjustment was
also made for CPU utilisation or load. We work under the assumption that the approximate workload would be similar immediately after a migration.

- **Information policy:** The opportunity cost algorithm makes use of the total memory in each node, the load average or percentage utilisation of each CPU and the maximum load of each machine in the cluster. The maximum load is used in order to determine the global maximum. The cost of each node is summed to give the current cluster cost. Migration is only initiated by the node with the highest current cost, a sender-initiated policy.

- **Selection policy:** The process to migrate is chosen by determining what the new cluster cost would be after the transfer. The process which reduces the overall cluster cost the most is chosen. If no process would reduce the cost, no transfer is made.

- **Location policy:** The location to transfer to is the one that currently has the lowest cluster cost, and as such the lowest workload. No attempt is made to transfer a process if there is little difference (we defined a constant of 0.1 for this comparison) between the sender’s cost and the receiver’s cost, since this indicates that the cluster is already fairly balanced.

### 3.3.3 Home model

Lavi and Barak’s [11] description of the home model describes two algorithms. We implement a variation on the AssignH algorithm, bearing in mind that it is defined in a job-centric fashion that our framework does not directly support. Our implementation determines the optimal workload for each node in the cluster as the average of all of the current load averages. We also added a condition which relaxed the allowable range of workloads a little. The paper suggests that the systems be balanced below $2 \times \text{optimal workload}$, and this works for clusters with 3 or more nodes, however some of our earlier testing used a 2-node system, so we added a workaround. If we detect a two-node cluster, we reduce this to $1.5 \times \text{optimal workload}$, as it is not possible for the first case to be triggered with only two nodes. All jobs are considered equally under this model with a single defined constant multiplicative factor (set to 1.05 during the experiments for a 5% penalty, chosen arbitrarily) representing the penalty of running a process away from its home node.

The paper also generalises the algorithm, allowing for the case where the optimal workload on the cluster is unknown. This is necessary if the nodes are
not fully connected. In our experiments they are always fully connected and as such we assert that the mean load average would be a reasonable optimal workload estimate and implement that instead.

- **Information policy:** The CPU load average of each node, distributed to all other nodes, is used by the algorithm. No other resources are considered by this algorithm.

- **Selection policy:** We migrate multiple processes until the workload decreases to below the $2 \times \text{optimal workload}$ level. Processes are chosen from the heaviest remote processes through the lightest local processes until the heaviest local processes on the node are reached. We define the weight of a process as the approximate proportion of the CPU it is currently consuming.

- **Location policy:** The location is chosen arbitrarily as the first node in our list that we believe would be able to accept the process and still remain close to the optimal workload (below the $2 \times \text{optimal workload}$ threshold).

### 3.3.4 Distributed MINIX load balancer

Our implementation of the distributed MINIX load balancer implements process migration of tasks already in progress, as with all of the algorithms. We are unable to track process arrivals, so all tasks execute for at least a short period of time on the initial machine. Our implementation continuously monitors the load values and checks for migration triggers from other nodes in the cluster.

- **Information policy:** This algorithm makes its decision based on the CPU usage percentage of all of the nodes, which is distributed across the network by the framework.

- **Selection policy:** The algorithm selects the process which is currently the most CPU-bound, or equally any remote process, since remote processes are maximally CPU-bound (a simplifying assumption). CPU-boundedness is determined by comparing the ratio $\frac{\Delta \text{user time}}{\Delta \text{user time} + \Delta \text{system time}}$ of the process with a pre-determined threshold. Remote processes under openMosix don’t report system time and as such get a ratio of 1.0 in this model.

- **Location policy:** The location chosen depends upon whether the migration was sender-initiated or receiver-initiated. In the sender-initiated case, we send the process to the first machine we find in our list that has a
low load. If no such machine is found, the migration does not occur. In the receiver-initiated case, the destination host is the source of the trigger message.

We follow the same specifications as Tsai et al. [15], defining a high load as greater than 75% CPU utilisation and a low load as less than 25% CPU utilisation. Our calculation of CPU-boundedness is different, making their threshold \( \frac{3}{4} \) by rearrangement of the formula. However we used 0.5 as experimentally this allowed processes to migrate more freely in our environment. This value could be optimised with further experimentation.

3.3.5 Memory ushering

Memory ushering is unique in the algorithms we selected in that it considers only the memory consumption of a node. In this way it could potentially be deployed alongside any of the algorithms whose workload is simply a function of CPU consumption (that is, any of the four previous algorithms with the exception of opportunity cost).

We implement Barak and Braverman’s [3] memory ushering algorithm in a very limited fashion. Since we do not provide easy access to the dirty page counts of each running process, we implement only the second stage of the algorithm. We count the number of resident pages in memory (the resident set size) as the size of the process.

- **Information policy**: The memory ushering algorithm uses the total memory in each node and each node’s percentage memory consumption that is distributed to all nodes. These are used to determine the free memory in each node. Migration is triggered when free memory drops below a predetermined threshold level.

- **Selection policy**: The largest process to fit in the node with the greatest free memory is chosen for transfer.

- **Location policy**: The location chosen is a two-step process. Firstly, the node with largest amount of free memory is chosen, which influences the process chosen for transfer. Then the nodes are re-examined and the node which would best fit the process is chosen.
In addition to developing a usable framework, we needed to test the algorithms that we implemented. We intended to show the effectiveness, in terms of execution time and amount of work done, of each of the algorithms, with the hope of identifying some characteristics of better algorithms. We also wanted to test the scalability of the framework and these algorithms, so we ran the tests across clusters of varying sizes.

4.1 Environment

We set up a cluster of 12 nodes using the process migration supplied by open-Mosix [2], running in the ClusterKnoppix [16] Linux environment. We disabled openMosix’s automatic load balancing during all of the experiments. To test scalability we used 4- and 8-node subsets of the 12-node cluster, modifying the openMosix configuration accordingly. To distribute statistics, the test harnesses connect to each other via a multicast UDP connection. An additional non-cluster machine recorded the statistics by listening in on the socket.

The test systems were identically configured Intel Pentium III 1GHz dual-processor machines with 1GB of RAM each. The console machine was a Dell laptop (an Intel Pentium III 733MHz with 512MB of RAM). All were booted using the same CD image, and therefore ran the same kernel and had identical environments. A USB mass storage device (a flash drive) was attached to the laptop for persistent storage of the statistics and logs.

The thirteen nodes were connected up via 100Mbit Ethernet to a single switch, implementing a fully-connected complete network. We ran a DHCP (Dynamic Host Configuration Protocol) server on the console machine to enable the other machines to boot automatically. We also added a SSH (Secure SHell) public/private key pair to the CD image to enable automatic remote logins to each machine. The built-in openMosix discovery daemon provided automatic config-
uration and membership of the cluster. A list of the modifications made to the ClusterKnoppix distribution is given in Appendix B.1.

We then ran benchmarks from the openMosix stress test [13] suite via shell scripting. The scripts called our test harnesses in turn and started the statistics capturing test harness on the console machine. The statistics captured were all of the statistics available to the algorithms for every node (that is, CPU consumption and load, and RAM consumption) We also collected the decisions made, the location of processes, and CPU and RAM consumption of the resource manager on each node by capturing the debugging messages output by each manager. Execution times were also recorded by the benchmark script.

The stress test suite was altered to allow each test to be executed independently. Further details on the modifications made are given in Appendix B.2. The tests we ran were:

- **Distkeygen**: Generates a large number of 1024-bit RSA key pairs in parallel processes.
- **Forkit**: Quickly creates many processes that perform computations within nested loops with large numbers of iterations.
- **Portfolio**: A stock market simulation written in the Perl interpreted language that splits itself into multiple processes.
- **Timewaster**: A test which runs frequent system calls on the clock in multiple parallel processes.

Each of these tests adjusts the amount of workload they perform depending upon the number of processors detected in the cluster at run-time, as is reported by openMosix. In most cases this is simply adjusting the number of parts that each task splits itself into, with each process performing the same amount of work. Therefore, in the 12-node tests more work is required than the comparable 4-node test and only limited comparisons can be made across tests with differing numbers of nodes.

We chose not to run the included LTP and LMBench tests. The LTP tests (from the Linux Testing Project) test the functionality rather than the performance of the system, and focus on the kernel. The kernels deployed across the cluster are all identical, and this sort of task would not be benefited by process migration since the system calls would continue to be executed on the home node. Similarly, the LMBench benchmarks are very time-consuming and are not designed for parallelism.

22
The purpose of these experiments is to identify characteristics that lead to more efficient load balancing across a cluster of machines. The tests each have differing profiles and will test different aspects of the algorithms.

We ran all of the stress tests with each of the load balancing algorithms. Additionally we ran the tests with no process migration at all to establish a baseline for comparison. We also compared the responsiveness of the Opportunity Cost algorithm implemented using the one-minute CPU load average against the same algorithm using the percentage CPU utilisation.

We also intended to look for any advantage to running the memory ushering algorithm, however this was difficult to test in the Live CD environment. When running from a Live CD there is no virtual memory should it become fully consumed, and therefore no room for error.

4.2 Results

We sampled and recorded the CPU load and memory statistics of each node and where processes were executing on the cluster once every second. This is the same information that was available to each algorithm.

To analyse the data we converted this information into a quantity we call the workload ratio of the cluster. The workload ratio is evaluated using Equation (4.1), where \( N \) is the number of nodes in the cluster, and \( T \) is the execution time of the test in seconds.

\[
workload\ ratio = \frac{\sum_{i=1}^{N} \sum_{t=1}^{T} CPU\% (i, t)}{100T} \tag{4.1}
\]

The workload ratio is a ratio of the percentage CPU consumption across the cluster and the execution time of the test. It is approximated, being derived from discrete sampled values. It includes all of the workload involved in the benchmark, in the algorithm (test harness) being tested, and arising from performing migrations. A workload ratio of 1 indicates that on average one node was fully consumed throughout the test, and a value of 10 would indicate that ten nodes worth of processing was being performed. The higher the number the better for workload ratio, with the theoretical maximum being equal to the number of nodes in the cluster. A value between zero and one suggests that a single system is not being fully consumed by the test, that is, it was left idle for some time.

Also important was comparing the execution times of the tests. In a real
situation we want not only load to be balanced but execution to remain efficient. The two sets of data are correlated (an increase in one decreases the other), but having both makes analysis of the tests easier and trends more obvious.

Graphed are the results of the 4-, 8- and 12-node experiments for each of the four benchmarks in figures 4.1 through 4.6. The four algorithms are graphed (including both variants of opportunity cost) alongside the baseline results. Note that some of the forkit tests failed and are not shown.

Graphing the raw data allows for a deeper analysis of the data. This can show the resource consumption across the cluster over time. This was useful in troubleshooting the behaviours of some of the algorithms. Such an example of these results is given in Figure 4.7. These show the difference in CPU consumption (one line per cluster node) throughout the duration of a single experiment set. This highlights the difference in migration behaviour.

For this test (distkeygen), we see that the home model implementation failed to do significant migration. Only one process appears to have been migrated to a different system. Looking at the logs directly, we can identify that more processes were migrated, but those processes were not contributing to the workload of the cluster. We also see that the opportunity cost implementation kept all nodes equally busy, as did the hydrodynamic implementation. The hydrodynamic went further than balancing though, in this case we see that the master node (192.168.0.1) actually becomes idle during the experiment. The distributed MINIX load balancer kept three nodes fully occupied but left one only half-busy. Remembering that the test machines are dual-processor machines, this suggests that only one of the distkeygen processes was transferred to that machine. We also see that distributed MINIX took a while to stabilise the CPU consumption after all of the jobs commenced, however this occurred at the same rate (about ten seconds) for each algorithm, excluding the home model.

In order to examine the scalability of the implementation, we also recorded the CPU consumption of the managers running on each node throughout each test. We show in Table 4.1, the results averaged over all nodes and benchmarks for each test and cluster size. This data is extended upon in the following graph (Figure 4.8), which adds the minimum and maximum average consumption.
Figure 4.1: The workload ratios for each benchmark/algorithm combination run on the 4-node cluster.

Figure 4.2: The execution times for each benchmark/algorithm combination run on the 4-node cluster.
Figure 4.3: The workload ratios for each benchmark/algorithm combination run on the 8-node cluster.

Figure 4.4: The execution times for each benchmark/algorithm combination run on the 8-node cluster.
Figure 4.5: The workload ratios for each benchmark/algorithm combination run on the 12-node cluster.

Figure 4.6: The execution times for each benchmark/algorithm combination run on the 12-node cluster.
Figure 4.7: Graphing the statistics over time is a useful diagnostic for the behaviour of the algorithms. Here we compare the four algorithms during the 4-node distkeygen experiments.
Figure 4.8: The CPU consumption of each algorithm over all of the tests is shown in this graph. The minimum, maximum and mean values in each data set are shown, with the relationship between the number of nodes highlighted.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CPU Load</th>
<th>Home model A</th>
<th>Distributed MINIX</th>
<th>Hydrodynamic</th>
<th>Opportunity cost CPU task 4</th>
<th>Opportunity cost CPU f 4 +</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-node</td>
<td>2.4</td>
<td>1.8</td>
<td>2.5</td>
<td>1.7</td>
<td>1.7</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>8-node</td>
<td>2.1</td>
<td>2.0</td>
<td>2.7</td>
<td>1.9</td>
<td>1.9</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>12-node</td>
<td>2.4</td>
<td>1.8</td>
<td>2.3</td>
<td>2.0</td>
<td>2.1</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 4.1: Average percentage of CPU consumed by resource management process while executing benchmarks in each category. Values are averaged over all nodes.
CHAPTER 5

Analysis

5.1 Comparing the algorithms

All of the algorithms tested managed to migrate processes, and in doing so, increased the efficiency of the cluster for most tests. The only algorithm which stood out as being less efficient was the home model implementation. We believe this to be down to our implementation of the algorithm, rather than the theory. It would perform better if it was able to track job arrivals and departures, which the framework does not directly provide.

In all cases, performance is reduced by having Linux initiate processes on a given node before we decide whether to move them or not. Ideally, the home model implementation would make its decision before the process was created (when the process request was made), however implementing this is left as future work, along with the future integration with the GOANNA system manager for DSTO.

5.1.1 Algorithm observations

A closer analysis of some of the data shows that the home model algorithm suffered from repeatedly attempting to migrate processes that openMosix would not migrate, and additionally from migrating processes that weren’t being executed (mostly background daemons). This affected all of the algorithms to varying degrees, because they operate under the assumption that all listed processes are running and can be migrated. It shows a dependence upon the ordering of processes, which is something where the home model differs significantly to the other implementations.

The two opportunity cost variants show the difference between two methods of determining CPU consumption, the methods being the one-minute UNIX load average and the ratio of idle CPU cycles to total CPU cycles (CPU %) in a
The hydrodynamic algorithm tends to shuffle processes around the cluster, as can be seen in the earlier Figure 4.7. This does not affect performance by much, but tends to cause the cluster to be more closely balanced not only in CPU utilisation (as indicated in the figure) but also with the CPU load average for some of the tests. This is due to the algorithm balancing the CPU load measure instead of the percentage CPU consumption used by some of the other algorithms.

The distributed MINIX algorithm had middle-of-the-road performance, working well across all tests. The benefit of distributed MINIX is seen when we consider the amount of CPU consumed by the resource management process itself later. On average it tends to consume less CPU than some of the other algorithms. We put this down to the algorithm being a simpler design than the others and relying upon less complicated data being kept.

5.1.2 Benchmark observations

Considering the benchmarks we ran, we see that all of the algorithms handled the portfolio benchmark very poorly. All of the execution times were longer than the baseline, and the workload ratios were rarely better than the baseline. However the baseline workload shows that the single node was hardly taxed by this test, suggesting that the algorithms may be being over-enthusiastic by performing as many migrations as they did. All other tests experienced a speedup from the algorithms in most cases.

The forkit test was a unique case, being the only benchmark where algorithms failed the test. This was due to the RAM consumption of the benchmark. We discovered that the master node, where all of the processes are started, becomes full. The test systems do not have access to any swap memory, and the RAM is also being consumed by a copy of the CD and a RAM disk. Some algorithms performed better than others under this constraint, being responsive enough to move processes away before the memory was filled and processes were terminated.

All tests completed on the 4-node cluster, however both the home model
implementation and the baseline test failed to complete on the 8-node cluster due to processes being killed by running out of memory. The fact that the baseline test failed shows that we were probably attempting to perform more computation than a single node could handle and confirms that the other tests were showing an improvement in capability as well as execution time. The opportunity cost variants were the only algorithms to complete this test for all 12 nodes. This is likely due to the inclusion of memory consumption in the calculations. The addition of memory ushering to the other algorithms may have enabled them to perform better in this benchmark.

5.2 Scalability

Scalability of the system is an important consideration. Here we have a testing environment which may or may not reflect the target environment and we need to identify how generic this solution is. For this reason we monitored the CPU and RAM consumption of the resource management process on each node during each of the tests.

Looking at the CPU consumption of the resource management processes on each node, (Figure 4.8), we see that the usage is variable depending on the algorithm, number of nodes and number of processes to be managed. In all cases the consumption is less than 10% of the capacity, averaging between 1 and 3 percent (Table 4.1). We noticed some spikes in the data sets, these may be occurring as processes terminate and new ones are started, however these usually last only one data point (or < 2 sec)). The master node, where all of the processes are started, generally has the greatest consumption (the maximum values shown in the graph), followed by the nodes that the algorithm chooses to migrate to most often.

Comparing the 4-, 8- and 12-node tests, we find that in some cases CPU consumption increases with the number of nodes (Figure 4.1). Both the averages and the maxima of the opportunity cost implementations show this trend. This suggests that these implementations would not be appropriate for a network of arbitrary size. The complexity of the calculations, which are performed on each node about each other node known, causes this problem for the opportunity cost algorithms.

As would be expected, the passive baseline values do not vary significantly in any of the tests.

The home model algorithms show the impact of not being able to successfully migrate processes. While the average CPU consumption remains low for the
home model (this is due to the other nodes in the cluster not having much work to do), the maximum average CPU consumption increases dramatically from 4 to 12 nodes. This maximum in all three cases occurs on the master node and is the result of managing and repetitively attempting to migrate multiple small processes away.

The distributed MINIX and hydrodynamic implementations show little change between the tests. The behaviour of the distributed MINIX algorithm makes more sense if we omit the outlier of the 8-node forkit test from the 8-node values (the next highest value of 3.3% fixes the trend). Then for both algorithms we see almost constant means and maxima. This suggests that these algorithms may be better suited to larger numbers of nodes. The hydrodynamic algorithm shows a decrease in the maximum average CPU consumption with increasing numbers of nodes. This can be explained by its job getting ‘easier’ with more neighbour nodes. This may actually be a side-effect of the chunking of workload problem, because with more neighbours the counters to each node increase more slowly and thus process transfers occur less frequently. However with more neighbours, the implementation must also perform more calculations to reach such a conclusion.

Removing the assumption of the framework that the network be complete and that each node know about every other node would enable larger networks to function more efficiently. As has been shown with the work on the hydrodynamic algorithm [6, 7], workload can be distributed a couple of hops away from its original node to balance the load. In larger networks, this can be more efficient because the amount of calculation required to be performed on each node is reduced.

It must be noted that the testing implementations we used output copious amounts of debugging information. This is not necessary for the operation of the resource manager, but is required for measurement during experiments. Reducing or eliminating this logging should also further reduce the CPU consumption of these processes.

As for memory consumption, which was also measured, the resource management processes were stable, using less than 1% (< 10MB) of each node’s RAM. This amount consumed varied little between algorithms, but increases marginally as the number of nodes is increased, due to the statistics for each node being stored in each process. Again, splitting a large network up (either virtually or physically) would reduce this cost.

Lastly, as the number of processors increased the apparent consumption of the cluster decreased. The best 4-node performance was 78% of the theoretical maximum, decreasing to 65% for 8 nodes and 60% for 12 nodes. This is another interesting note about the scalability of the system. This may be a consequence
of the chunking of workload into processes, in which case having more running processes would likely improve this figure.

Network bandwidth consumption is another important measure for scalability. We chose not to measure this statistic because it was difficult to implement and because our framework already implements efficient multicast traffic. Multicast traffic has the property of having each packet sent only once over each network segment irrespective of the number of receivers (ideally greater than one) on that segment. (Both the hydrodynamic and distributed MINIX implementations have an additional unicast communications channel that is very low bandwidth and does not affect this assumption.) The other major sources of network traffic are process migrations and communications from remote processes. These are implemented by openMosix in our experiments, which is not intended to be part of the end product, and as such it was not analysed here.

5.3 Further observations

As seen above, at no time does the resource manager fully consume the cluster. The workload ratio is often significantly less than the number of processors being exploited. This value is an average taken over the length of the test, so this may be unrepresentative of reality. It is likely a side-effect of chunking — that is, we can only move about baskets of workload (individual processes) rather than atomic amounts of work. Having larger numbers of parallelizable processes makes it easier for the algorithms to balance the cluster than few ‘chunky’ processes. This can be seen in the results of the distkeygen and timewaster tests that split themselves into multiple processes each performing a very similar task. As a result it is these benchmarks that achieve the highest workload ratios.

The results do confirm that process migration offers benefits in many cases. It can significantly improve the performance of connected machines. Consider the observations for the baseline case, where in three out of the four benchmarks we see execution times reduced. As the amount of workload per process and number of processes increases, such as with the timewaster and distkeygen tests, the benefit increases dramatically.

We did not consider varying the update rate and how often the algorithms were run. Potentially optimising these values could lead to a stable load balancing solution with better workload ratios. In addition, there are some constants in the definition of some of the implementations (consider the threshold value for the hydrodynamic algorithm as an example) that may also benefit from optimisation.
CHAPTER 6

Conclusions

The results suggest that for load balancing algorithms there does not appear to be any model that suits all cases. The best-performing methods are those that optimise both CPU and RAM usage, such as the opportunity cost approach. However, there is a trade-off between optimisation of the cluster and the CPU time spent in the manager. This suggests that simpler approaches will often be more appropriate than the more intensive approaches. We also saw that algorithms are very sensitive to the selection of processes, and are heavily dependent upon the underlying process migration mechanism for task transfer.

The metrics that are used to determine the consumption of a resource are also important, as we saw in the comparison of the two opportunity cost variants. For the improved responsiveness it gave, we saw that percentage consumption was better than the simple one-minute UNIX load average.

The framework that we developed will be useful in continuing to analyse better models. It can be extended to support other metrics of workload, additional resources, different migration techniques or different algorithms. Along with the suite of test harnesses built up so far, performance of a cluster with different types of tasks running could be analysed. The framework as it stands is suitable for moderate sized clusters, however since each node is required to process the statistics of every other node, modification may be required to extend this work to larger clusters.

Along with integrating the framework into DSTO’s GOANNA architecture, future work to benefit this area would be to try to implement the job-centric approach. That is, being able to trigger the algorithms before a process determines its home node. This would enable an improved implementation of the home model algorithms. It would also be beneficial to remove the assumption that a network be complete and fully connected from the framework.

Additionally more work into optimising the framework for the application domain would be beneficial, by way of optimising some of the constants used by the algorithms, and incorporating more robust communications between nodes.
APPENDIX A

Original Honours Proposal

Title: Resource Management in the GOANNA Computing Environment

Author: Joshua King

Co-supervisor: Dr. Nick Spadaccini

Co-supervisor: Dr. Gary Bundell

CEED Mentor: Michael Strickland, DSTO Stirling

Revised: 22 May 2005

A.1 Background

The Generic Open Architecture for New Naval Applications (GOANNA) computing environment is a system architecture design for distributed systems. The environment is designed to host multiple independently developed software components. The components are generally either multi-threaded C or C++ applications, or distributed Message Passing Interface (MPI) applications. This architecture is currently under active development by the Defence Science and Technology Organisation (DSTO) on the HMAS Stirling naval base.

The architecture uses open standards and commercial off the shelf hardware to create a low cost, high performance cluster environment. An important consideration of cluster and parallel-processing environments is how the limited resources, such as CPU time and memory, are shared among the different components. This problem of resource management is important to maximise the efficiency of the cluster, especially in a real-time context.

Resource management can be split into static and dynamic resource management. Static resource management is those decisions made to allocate resources at the beginning of the execution of components on the cluster. Dynamic resource management is monitoring the actual resource usage and making decisions whether to move processes while they are executing. Algorithms determine
when and how to move processes, which influences the performance, in terms of efficiency or utilisation, of the cluster.

The GOANNA architecture provides for a Resource Manager component whose role is to monitor the resource usage of each node on the cluster. One of the Resource Manager is a co-ordinator and handles making decisions about migrating components around the cluster and where to introduce new components. The actual migration of processes is performed by the System Manager component that the Resource Manager reports to.

A review has previously been done for DSTO as to some toolkits for resource management, and this material may be drawn upon in this project.

A.2 Aim

This project aims to discover algorithms that can deliver the most efficient use of a cluster based on the GOANNA model. Both static and dynamic resource management are to be investigated. Currently implemented algorithms will be researched and tested. This will lead to an improved implementation of resource management for DSTOs GOANNA architecture. Any results may also be extended to other cluster implementations. The Resource Manager component will be developed to include these algorithms for DSTOs use.

A.3 Method

The project will encompass the following tasks (taken from the CEED project document):

1. Research algorithms that can be used to distribute components amongst the nodes in the GOANNA computing environment in a manner that ensures each component receives the resources it needs whilst minimising the need to migrate components between nodes and maintaining acceptable system performance.

2. Write software that implements the resource management algorithms. This software will run as a stand-alone executable (Resource Manager) in the GOANNA computing environment, and will communicate with the GOANNA System Manager via an interface that has been designed for this purpose.
Algorithms will be researched for in existing products and literature. This will be the first part of the project.

Subsequent to this research, the software can be implemented. It is to be designed so that algorithms may be inserted into it as they are implemented. In this way different algorithms may be compared experimentally as necessary to determine their relative effectiveness.

Further research can then be performed alongside implementation once the initial framework has been completed.

This project is being conducted under a modified CEED project model, involving intensive periods on-site at DSTO as well as work at university. This leads itself to the following timetable:

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semester 1</strong></td>
<td>Research existing algorithms</td>
</tr>
<tr>
<td><strong>Inter-semester break</strong></td>
<td>Implement resource management system and test</td>
</tr>
<tr>
<td><strong>Semester 2</strong></td>
<td>Analyse results of test, try to suggest better algorithms</td>
</tr>
</tbody>
</table>

A thesis, seminar and poster will be prepared alongside these activities.

A.4 Software and Hardware Requirements

For experimentation, an existing computing cluster will be used at DSTO Stirling. Otherwise for the most part only standard computing equipment is required for this project.

Most of the standards to be investigated are freely available, and the research materials will be accessed through the UWA or DSTO research libraries.
APPENDIX B

Test Environment

We modified the stock versions of ClusterKnoppix [16] and the openMosix stress test suite [13] to suit the needs and limited time of our experiments. The location and versions of these tools is given in the Bibliography, and our modifications are given below.

B.1 Modifications to ClusterKnoppix

We made the following modifications to ClusterKnoppix [16]:

- Made the console runlevel 2 the default.
- Removed most of the graphical applications from the CD.
- Added the OpenSSH server to the default startup.
- Created an SSH key pair and corresponding entries in the `/root/.ssh/known_hosts` file to enable automatic remote login.
- Disabled console screen blanking at startup.
- Changed the ISOLinux configuration to automatically boot and load the CD into RAM, enabling the CD to be removed without user interaction.
- Added the openMosix stress test and our test scripts to the CD.

We made extensive use of the Knoppix Remastering Howto [9] in order to make these modifications.
B.2 Modifications to openMosix test suite

We made the following modifications to the openMosix test suite [13]:

- Applied the patch for the Portfolio program listed on the test suite web site.

- Changed distkeygen (distkeygen/distkeygen.c) to allow specification of the number of keys to generate on the command line (this was done to reduce execution time and memory consumption of the benchmark).

- Changed the benchmark script (start_openMosix_test.sh) to allow the selection of a particular test to execute on the command line (otherwise all tests are executed as before), to work with changes to distkeygen, to disable the moving test, to fix an issue with the report file naming, and to allow the test suite to exist in locations other than the default location.

- Modified the number of iterations in the loops making up the timewaster and forkit benchmarks.
Bibliography


