Resource Based Policies for Load Distribution

Kristian Paul Bubendorfer

A thesis

Submitted to the Victoria University of Wellington
in partial fulfilment of the requirements for the degree of
Master of Science with Honours in Computer Science.

Victoria University of Wellington

August 31, 1996

To Andrea

Abstract

The sharing of computing resources in a distributed system is a more complex matter than in an equivalent centralised system, due to the fragmentation of resources over a set of autonomous and often physically separate hosts.

The effect of this fragmentation is reduced by evenly distributing the workload of the system over all the hosts, resulting in more effective use of the system resources, and a corresponding reduction in the average response time of processes.

This thesis concentrates on deciding how the system workload may be assigned to each host, based on the resources currently available at a host, and the resources required by the processes being distributed. Two different approaches are suggested, based on initial placement and process migration (distributing processes before and while they execute respectively). These approaches are evaluated using a trace driven distributed system simulator.

To distribute processes with initial placement requires knowledge of a process's resource requirements before it executes. Therefore the use of a simple form of prediction to provide this *priori* information is explored.

The major findings of this investigation are, that assigning workload to hosts based on their respective resources is worthwhile, and that process migration offers no distinct performance advantage over initial placement.

Acknowledgements

I would like to express my thanks to my supervisor John Hine, for his guiding hand; Paul Martin, for his encouragement and helpful comments, and to my wife Andrea, for reading and improving endless drafts.

Thanks also to Stuart Marshall and Ben Wong for being guinea pigs, and to Pinin Farina and Alec Issigonis for providing an outside interest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of distributed computing systems together with the corresponding movement from traditional centralised systems, has increased the difficulty in the sharing of system resources. The difficulty arises as the system resources can no longer be placed under the local control of a single agent operating on a single machine, but instead are fragmented between smaller, independent, and often physically separate systems. This fragmentation is a serious problem, as contention for resources now occurs on a host by host rather than on a system wide basis.

Early research identified, that if each host in the distributed system receives a similar amount of workload, then the effect of the resource fragmentation is reduced. This redistribution of the system workload is termed *load distribution*. Load distribution may also be desirable for other reasons, such as the reduction of remote communication, increased due to the fragmentation of the workload. Hence the *policy* of what to distribute and why, is distinct from the *mechanism* which physically distributes the workload.

1.1 Motivation

Early work by Eager et al. [ELZ86b] determined that simple load distribution policies, such as randomly selecting a host, are almost as effective as their more complex counterparts. As simpler policies require less state, they concluded there was little to be gained from pursuing more complex policies. Consequently, policies have often been neglected in the study of load distribution, being dismissed as trivial and discarded for the more tangible aspects of the load distribution mechanism.

Nonetheless, research into load distribution policies has continued, much of it concentrating in areas with different granularities (such as threads and objects), differing communication paradigms (remote object invocation) and heterogeneous systems, where the work by Eager et al. does not apply. The work that has continued with conventional processes and homogeneous systems, has tended to focus on extending the simple algorithms, training learning mechanisms, or more rarely, using *a priori* information to distribute the workload.

1.1.1 Goals

The use of prediction to provide *a priori* information on job resource requirements, has so far received only a small amount of research attention (see chapter 4), yet that work has been sufficient to demonstrate the approach is feasible. To the best of my knowledge, the only previous research to apply this form of *a priori* knowledge to distribute the load on a homogeneous system is Goswami et al. [GDI93].

The purpose of this thesis is to explore the possibilities for scheduling workload based on *a priori* information. In particular, to:

- Design a dynamic load distribution technique for scheduling jobs and processes to hosts, based on resource requirements and availability.
- Explore the use of dynamic *a priori* information for initial placement.
- Investigate the quality of prediction required for providing *a priori* information in load distribution.
- Determine if initial placement is comparable to process migration when *a priori* information is used in placement decisions.
- Investigate if policies which use both initial placement and process migration perform better than those relying on only one.

These goals are investigated using a trace driven distributed system simulator.

1.2 Thesis Organisation

The thesis follows a course, from a review of existing work, through to the presentation of the results of the investigation. The purpose of this section is to provide a brief overview of the chapters.

- **Overview of Load Distribution:**

  This chapter provides a general review of load distribution, aided by the introduction of a taxonomy for comparing, and constructing load distribution schemes.

- **Balanced Resource Allocation:**

  This chapter introduces and argues for the *process mix* hypothesis, which determines how processes should be allocated to hosts. Extensions to the *process mix* hypothesis are also presented as additional points of investigation.

- **Related Work:**

  With the objectives of the thesis clear, this chapter presents work that is either similar in intention or has in some other way dealt with similar problems.

- **A Testbed:**

  The *process mix* and associated hypotheses need to be evaluated before any claims can be made about their validity. This chapter defines the experimental testbed simulator on which evidence for the support of the hypotheses is measured. The testbed is comprised of three separate modules, the simulated distributed system, the mechanism for load distribution and the algorithms which test the hypotheses.

- **Distributed System Simulation:**

  The distributed system simulator seeks to emulate a physical system, on which jobs and processes can be executed. The purpose of this chapter is to detail the models and assumptions used in the construction of the simulator.

- **Load Distribution Mechanism:**

  The focus of this thesis lies with the load distribution policies rather than in the physical means of implementing the load distribution decisions. This chapter describes a simple load distribution mechanism that is used for all load distribution over the simulated distributed system.

- **Load Distribution Algorithms:**

  This is one of the most significant chapters in the thesis. It includes descriptions of all of the load distribution policies, and the algorithms that implement them. The most important aspect is the introduction of the algorithms which attempt to maintain the *process mix* of the system.

- **Experimental Work:**

  This section details the methodology and experimental results obtained in the investigation of the *process mix* and associated hypotheses.

- **Conclusions and Future work:**

  The results indicate there is merit in the use of *a priori* information to perform resource based load distribution in homogeneous systems. This chapter firstly provides some direction to future work, and then finally presents the conclusions of the thesis with regard to more general issues of load distribution.

- **Appendices A-D:**

  These appendices contain significant graphs summarising the performance of the various load distribution algorithms.

1.3 Contributions

This thesis contributes to the current load distribution research in at least three main areas:

- A load distribution policy is presented that maintains the balance of a host's resources and the workload over the distributed system. This policy is applied to both initial placement and process migration systems and evaluated on a trace driven simulated distributed system. The results demonstrate that this policy improves the performance of the system with both initial placement and process migration.

- The performance of the initial placement version shows that more complex initial placement policies can achieve useful performance gain over simple policies and are worth more consideration. This contrasts the earlier findings of Eager et al. [ELZ86b].

- Process migration is shown to perform no better than resource based initial placement, confirming the findings of Eager et al. [ELZ88], in contrast to the analysis of Downey and Harchol-Balter [DHB95], who question the validity of the modelling and its applicability to modern systems.

Part I

Chapter 2

Overview of Load Distribution

This chapter seeks first to introduce load distribution as a concept, and then as an active field of research. Section 2.2 describes dynamic load distribution in general terms. A taxonomy is presented in section 2.2.3, enabling load distribution policies to be summarised and compared.

2.1 Models of Computation

The general models of distributed[1] computation will be described before returning to discuss load distribution in detail. The particular model of computation can influence the choice and design of a load distribution scheme, and as such, requires clarification.

In order to place distributed computation in its correct context, a short section on the centralised model is included. The discussion will not extend to parallel architectures[2].

2.1.1 Centralised System Model

The centralised system consists of a set of terminals connected to a single central computer. The central computer is typically a timesharing system which switches frequently between processes to provide a share of the system resources to each. In this way, the timesharing system can provide an interactive environment for all users. The major drawback of this scheme is the need for a very powerful central computer. The primary advantage is that the entire capacity of the central computer is available.

[1]For loosely coupled systems. Tightly coupled systems, such as those sharing a clock and memory between CPUs, are outside the scope of this thesis.

[2]MPP, SIMD, dataflow etc.

2.1.2 Distributed System Models

Distributed computing owes its existence to two technological advances: the evolution of cheap powerful CPUs, and the advent of high speed networks with which to interconnect them.

Distributed systems provide a number of features over the single CPU timesharing system:

- **Resource Sharing:** Resources available at another site can be harnessed.
- **Concurrency:** A single computation may be able to exploit its parallelism by executing subparts in parallel on different hosts.
- **Reliability:** The system is robust with respect to single points of failure.

Hence the motivation for distributed systems is high, but at the cost of administration, and the underutilisation and redundancy of computing resources. In centralised timesharing systems, the entire computing resources of the system are available to be shared between all active processes. In a distributed system however, the CPU power is fragmented over a number of different hosts, which generally means a process can only access the computation resources of the host on which it is executing. This means that if more processes are executing on one host than another, each process is not getting a fair share of the system resources.

The Workstation Model

The workstation model is a group of personal workstations, connected by a high-speed network, illustrated in figure 2.1. This model has a number of variations in which the workstations can be peers[3] or disparate[4]. Additional information on this model can be found in Tanenbaum [Tan92], pages 524-530, which discusses the variations in detail.

As workstations are usually sited on desks and in offices rather than in a machine room, there is an element of 'ownership' associated with individual hosts [DO91], rather than a notion of a pool of shared resources. This viewpoint has important ramifications in load distribution and is discussed in detail in section 2.3.1.

[3] Of equivalent status, namely no particular workstation is designated as a file or compute server.
[4] Where there may be particular hosts with special functions, such as, a file server.

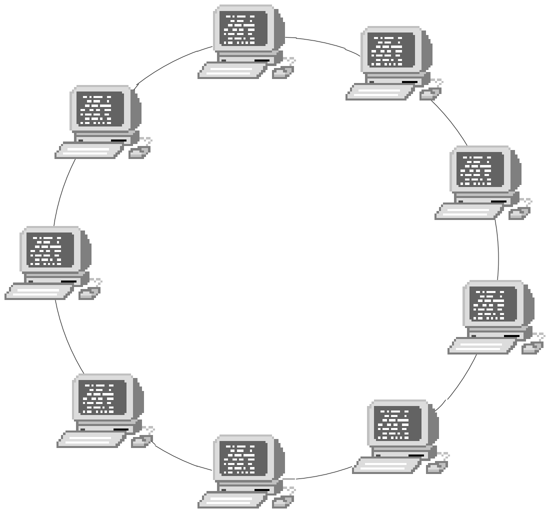


Figure 2.1: *The workstation model features a collection of independent workstations connected by a network.*

The Xterminal-Server Model

The Xterminal-Server model is a hybrid of the workstation and timesharing models, as shown in figure 2.2. Essentially, instead of a single central computer, there is a comparatively small set of compute servers forming a distributed system. Each compute server is directly responsible for a set of Xterminals, but any compute server in the system can be used by the user at any Xterminal.

This is the model of distributed system in use at the Computer Science Department at Victoria University, and it is this system from which all workload traces were recorded. As a consequence, it is also the model used in the testbed simulation, and in the design of the load distribution policies.

The Processor Pool

The processor pool model is another form of distributed system developed to exploit the availability of small cheap microprocessors. Amoeba is an example of such a system and is described in Tanenbaum et al. [TvRvS^{+}90]. The processor pool model can also be viewed as a form of the older timesharing model, section 2.1.1, with the single large compute server replaced by a large pool of CPUs. This system uses Xterminals to interact with the processor pool and allocates processors to users as they are required,

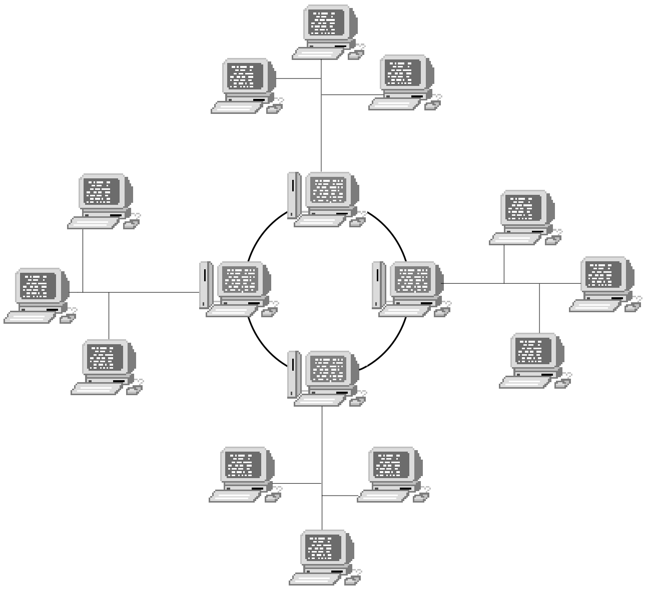


Figure 2.2: *The Xterminal-Server model consists of a set of compute servers each of which is responsible for a set of Xterminals.*

figure 2.3.

Processor allocation strategies are wide and varied, and include the potential for load distribution. The major difference between this model and the Xterminal-Server model is the number, power and autonomy of the processors involved.

2.1.3 The Granularity of Computation

Computation within a distributed system can take place with varying degrees of granularity, (granularity being the size of the unit of work within the system). In particular, different distributed operating systems may typically support any of the following granularities (listed in increasing size):

- Instructions
- Remote Procedure Call (RPC)
- Threads
- Objects
- Processes

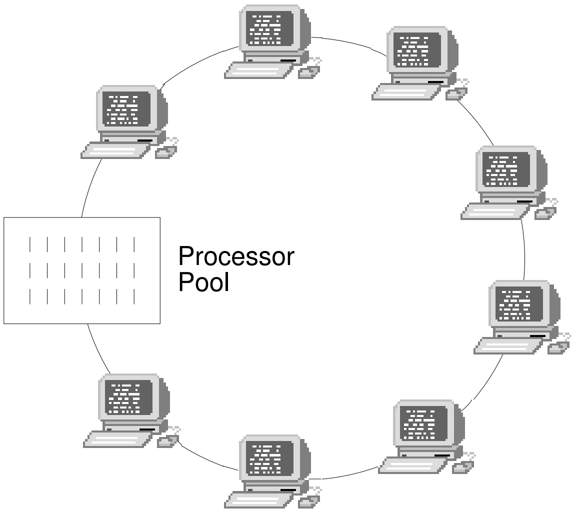


Figure 2.3: *The processor pool consists of Xterminals served by a pool of processors. The processors are allocated and balanced as required by the workload.*

- Groups of related processes

The smallest grain size is the instruction, and is only feasible in tightly coupled systems or architectures such as dataflow [GKW85]. The larger granularities are used in a variety of more loosely coupled operating systems, and it is these that are of particular interest.

A simple type of load distribution with RPC is possible by selecting between multiple hosts providing the same RPC service. Threads on the other hand, are not immediately applicable to load distribution due to the shared nature of thread address spaces. In the Mach 3.0 system, Ford and Lepreau [FL93], migrate the calling thread and use it to execute the code on the RPC server, but this is restricted to local[5] RPC only.

The Chorus Object Oriented Layer (COOL) [AJJ+92] emulates a distributed shared address space across a set of hosts, and could address the problem with migrating threads from shared address spaces, however COOL uses an object abstraction, and migrates objects rather than threads between hosts.

The group is a less common unit of distribution, of which a prominent example is a prototype of the V system. Theimer et al. [TLC85] migrated virtual hosts, containing a process and all of its children, between physical hosts. Multiple virtual hosts could reside on each physical host, but a virtual host could not reside on more than one physical host, hence the need for migrating the entire group.

[5]RPC when the client and server coexist on the same host.

The granularity of computation that I will use is the process. Unstarted processes are referred to as jobs, and both processes and jobs are distributable.

2.2 Dynamic Load Distribution

Load distribution seeks to improve the performance of a distributed system, usually in terms of response time or resource availability, by allocating workload amongst a set of cooperating hosts.

This division of system load can take place *statically* or *dynamically*:

- *Static load distribution* assigns jobs to hosts probabilistically or deterministically, without consideration of runtime events. This approach is both simple and effective when the workload can be accurately characterised and where the scheduler is *pervasive*, in control of all activity, or is at least aware of a consistent background over which it makes its own distribution. Problems arise when the background load is liable to fluctuations, or there are jobs outside the control of the static load distributer.

- *Dynamic load distribution* is designed to overcome the problems of unknown or uncharacterisable[6] workloads, non-pervasive scheduling[7] and runtime variation (any situation where the availability of hosts, the composition of the workload or the interaction of human beings can alter resource requirements or availability). Dynamic load distribution systems typically monitor the workload and hosts for any factors that may affect the choice of the most appropriate assignment and distribute jobs accordingly. This very difference between static and dynamic forms of load distribution, is the source of the power and interest in dynamic load distribution.

The objectives of this thesis lie entirely within the domain of dynamic load balancing. For brevity, I will take the more general term of load distribution to stipulate only the dynamic form.

[6]A batch process that will with probability P sort a 10 element file and with a probability of (1-P) sort a 100000000 element file is difficult to characterise, and the actual behaviour can only be determined at runtime.

[7]Where there is workload that is not scheduled, or scheduled by independent agents.

2.2.1 The Degree of Load Distribution

The essential objective of load distribution is the division of workload amongst a cooperating group of hosts. This objective may be fulfilled with varying degrees of fineness, the exact choice of which, depends on the environment and architecture of the system in question.

Load distribution is usually described in the literature, as either load balancing or load sharing. These terms are often used interchangeably, but can also attract quite distinct definitions. I will adopt the two terms, and use them in the strictest sense to describe the degree to which workload is distributed, and introduce a third term to describe the middle ground.

- **Load Sharing:** This is the coarsest form of load distribution. Load may only be placed on idle hosts, and can be viewed as binary, where a host is either idle or busy.

- **Load Balancing:** Where load sharing is the coarsest form of load distribution, load balancing is the finest. Load balancing attempts to ensure that the workload on each host is within a small degree (or balance criterion) of the workload present on every other host in the system.

- **Load Levelling:** Load levelling occupies the ground between the two extremes of load sharing and load balancing. Rather than trying to obtain a strictly even distribution of load across all hosts, or simply utilising idle hosts, load levelling seeks to avoid congestion on any one host.

Load sharing systems are common, and include Sprite [DO91], Butler [Nic87] and V [TH91]. The distinction between load levelling and load balancing schemes is more difficult. In particular, to meet the definition, a load balancing scheme must continue to redistribute load until it meets a balance criterion. One such scheme from Kara [Kar94] only considers the system balanced if the difference between the least and most loaded hosts is within a limit Δ. Other schemes such as, MOSIX [BS85], which could be considered load balancing systems, are in fact load levelling, as the balancing phase occurs periodically.

The architecture of the system is important, as it can suggest the most appropriate degree of load distribution. For example, a large network of personal workstations would be prohibitively expensive to balance[8] due to the overheads of load and state collection, yet the detection and utilisation of idle workstations for load sharing is quite feasible as shown by the Butler system [Nic87].

Thus load sharing, levelling and balancing define a continuum from a coarse to a fine distribution of load, and seek to distinguish the sometimes unstated intentions of different load distribution schemes.

2.2.2 Previous Load Distribution Taxonomies

There are numerous existing taxonomies available for the classification of load distribution, including Wang and Morris [WM85], Casavant and Kuhl [CK88] and Jacqmot and Milgrom [JM93]. Wang and Morris proposed a simple classification based on whether the load distribution was source or receiver initiated, and the degree of information required by the policies. This classification is too coarse however, and does not give a sufficiently detailed means of comparison between load distribution policies that are similar in one or both of the classes. In particular, most current distribution techniques are sender initiated, and therefore this classification provides little distinction between the members of this class. Other problems arise when the symmetrically initiated schemes are considered, in these cases load distribution schemes are both sender and receiver initiated and cannot be accommodated by the taxonomy. Casavant and Kuhl provide a broad classification scheme that includes both local and global scheduling, but this also did not facilitate the detailed comparison of algorithms within each class. Jacqmot and Milgrom had an entirely different approach, where it was the order in which load distribution decisions were made that was significant, for example, they identified four sequences:

- No explicit identification of the candidate process.
- Identification of the candidate process precedes identification of the target host.
- Identification of the target host precedes identification of the candidate process.

[8] Balancing within clusters and amongst the clusters in a hierarchical fashion is a partial solution.

- Identification of the source host precedes identification of the candidate process.

This taxonomy does not however, encourage comparison or identification of the components of a load distribution scheme. It also cannot classify symmetrically initiated policies, or distinguish between initial placement and process migration.

2.2.3 A Load Distribution Taxonomy

The load distribution taxonomies mentioned in the previous section all fall short of providing a means for clearly discussing and summarising load distribution schemes. The taxonomy presented in this section concentrates on the individual decisions made during load distribution and therefore identifies the separate components of a load distribution scheme.

Load distribution schemes are usually divided into the *policy* and the *mechanism*. The policy is the set of choices that are made to distribute the load. The mechanism carries out the physical distribution of load and provides any information required by the policies (a useful analogy is the separation of the mind and body).

The division of policy and mechanism can be continued, breaking any load distribution scheme into a set of distinct but interdependent *components*. Figure 2.4 illustrates a decomposition suitable for the classification requirements of this thesis, with each leaf representing a distinct component of a load distribution scheme. The emphasis is on the components of the policy, and the provision of information to the policy by the mechanism.

The following list briefly explains each component identified by a leaf of the taxonomy. A more detailed look at each of the components is in section 2.3 where existing systems are used to illustrate the application of the taxonomy.

- **Participation Policy:** The participation policy determines whether a host will take part in load distribution. It can be as simple as a threshold based on the local load, or a more complex security issue such as the refusal of an intended migration from a untrusted server.

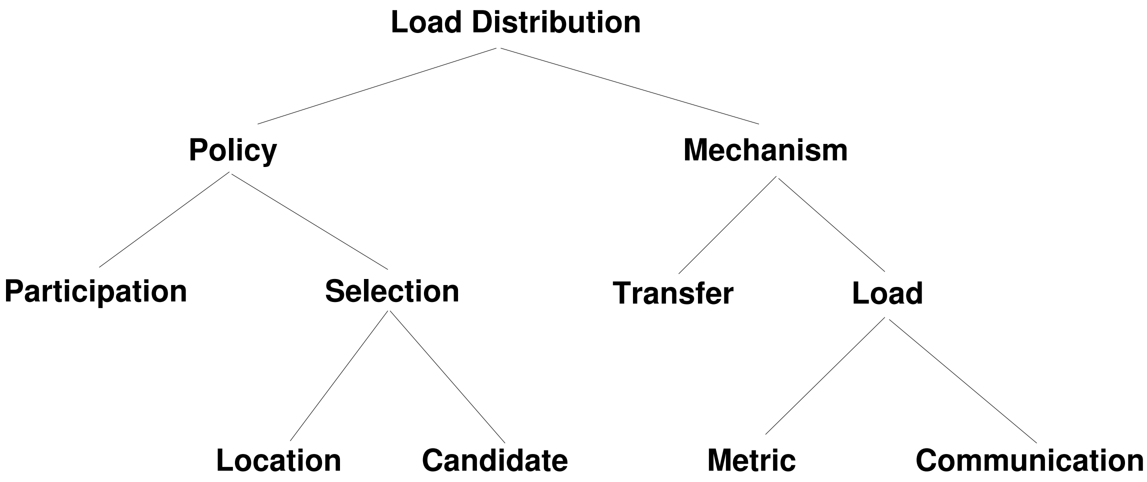


Figure 2.4: *The components of a load distribution system*

- **Location Selection Policy:** The location policy is responsible for selecting the participating hosts between which load is distributed. This policy is also responsible for handling the homogeneity or heterogeneity of the participants.

- **Candidate Selection Policy:** The candidate policy selects the jobs, process, objects, or whatever the mobile workload unit is, to be distributed. This can be for reasons ranging from insufficient service on the current host, to the reduction of communication paths.

- **Load Metric Mechanism:** The load metric is the representation used to describe the load on a server, and also in this thesis, the load imposed by a candidate on its host.

- **Load Communication Mechanism:** Load communication defines the method by which information, such as the load on a host, is communicated between the host and the load distribution policies. The load communication policy can also include the communication between cooperating distributed policies.

- **Transfer Mechanism:** The transfer mechanism is the physical medium for transferring jobs or processes between hosts. This leaf can be further expanded to include branches for initial placement and process migration.

The terms used in this taxonomy appear frequently in the literature, often with different and inconsistent definitions. The only meanings attached to these terms in this thesis are the ones outlined above.

In particular, the participation policy seems to be yet another name for a concept that is already well described, Eager et al. [ELZ86b] refer to the participation as a transfer policy, as does Zhou [Zho87], while Kremien and Kramer [KK92] describe it as an acceptance policy. However, the transfer policy defined by Eager et al. and Zhou combines the functions of the candidate selection and participation policies and is a coarser component. The acceptance policy of Kremien and Kramer is also different, as it defines the simple acceptance or refusal of incoming load, and does not include the initiation of load distribution.

Some of the categorisations also differ, for example Kremien and Kramer as well as Zhou, suggest an *information policy* rather than treating it as part of the mechanism. This definition however, contradicts my earlier distinction between policy and mechanism.

Another caveat in some load distribution schemes is that some components identified in the taxonomy are combined, while others may be completely ignored. A purely random policy for example, has no load communication mechanism and it's candidate selection policy is combined with the participation policy. The combination arises as all jobs arriving at a participating host (sender) are eligible by the fact of participation.

Component Interdependence

Although the leaves in the taxonomy are distinct, the individual components are not independent. The choice of load metric for example, affects the location selection and possibly the candidate selection as well.

If this taxonomy were used to construct a load distribution mechanism bottom up, by selecting components and joining them together, or to identify and then replace a single component, care must be taken to ensure that each part of the system is capable of providing the functionality required by the rest of the system. As an example, consider the mismatch in a system where a candidate selection policy required migration to reduce communication paths, but was provided with initial placement as the transfer mechanism.

2.3 Design Approaches

Each component of a load distribution scheme represented in the taxonomy could be dependent on the system for which the load distribution is intended, or may simply be a choice made for performance reasons or personal preference. All can however, exhibit a wide variety of interesting approaches and unique solutions due to the large design space.

This section is intended both as an overview of the general field of load balancing, and as an exercise of the taxonomy.

2.3.1 Participation Policy

The participation policy decides whether any particular host should be involved in a load distribution activity. The reasons for participation can be varied and some approaches are detailed in the following subsections.

Threshold

A two level threshold policy has a lower threshold that defines at which point a host may begin to initiate load distribution to offload any excess load. An upper threshold determines the point at which an overloaded host may start to refuse incoming foreign load[9].

This gives two levels of participation: a restriction on the minimum amount of local load before any is offloaded, and a symmetric restriction on the minimum amount of local load before incoming load can be refused. Outside these limits a host may only participate as either a receiver or source of load. One example of this approach is Holzer[Hol96].

Ownership

Sprite [DO91] has an interesting participation policy, based on the concept of ownership. A fundamental view of the workstation model is that each workstation is owned by the user currently logged in from the console - whom I shall call the *primary user*. This concept of ownership is considered absolute, and Sprite immediately evicts all foreign processes when activity from the primary user is detected. The intention of this eviction

[9] Foreign load is load that is offloaded by other hosts onto the host.

policy is to encourage primary users to share their personal workstation, with little detriment to their own work.

Thus the participation policy is: only hosts that are considered idle (as described in section 7.1), may be selected as a destination for load distribution. This policy goes even further, in that any foreign processes must be migrated from the current host on detection of an active primary user.

The V system described in Theimer et al. [TLC85] also preserves the concept of ownership, but requires the user to manually initiate the eviction of foreign processes.

Another system supporting ownership rights is Butler [Nic87]. Butler does not support migration, and as a consequence, foreign processes are terminated on detection of an active primary user. This rather ruthless policy was seen as the only option in a system with single user workstations and without migration.

Security

Security can also be a factor in any participation policy. Powell and Miller [PM83] suggest hosts could refuse to accept a migration request if the source host is not trusted, such as one from a different administrative domain.

2.3.2 Location Selection Policy

The location policy is responsible for selecting the source and or destination hosts between which load is to be transferred. There are four main approaches to determining the location of source and destination hosts.

Centralised - Dictatorial

In a *centralised* location policy, a single agent is responsible for deciding the source and destination hosts. This system uses load values reported to the agent to determine say, the least and most loaded hosts and initiate a load transfer between them. A prime example of the centralised approach is Condor [BLL92], which is based on the workstation model and therefore has ownership constraints.

In Condor, there is a single central agent and a local agent on each host. The central agent is responsible for sharing the resources of the system, and when it discovers an idle host B, it decides which host A may execute processes on it. The central agent then contacts the local agent on A and gives it permission to execute processes on B. After this point the local agents on A and B negotiate the transfer of processes.

The participation of host B is not delegated to the central agent as host B may still refuse a transfer from host A, if the user has returned.

The *dictatorial* approach is used by Harchol-Balter and Downey [HBD95] in their simulated load distribution scheme. Here a centralised agent periodically identifies the most and least loaded host and orders a migration between them. There are no ownership constraints, and the only way a migration is avoided is if there are no suitable candidate processes on the source host.

In dictatorial policies such as this, it is likely that the participation policy is also delegated to the central agent, which would have to resolve security issues and participation thresholds.

Sender Initiated

Sender initiated is the most common form of load distribution [Ber85, Rub87, DO91], and occurs when a source host decides to participate in load distribution, perhaps due to its exceeding a local load threshold. In this case, the source host is fixed and the location policy is only required to identify a destination host which will participate. The actual policy may be to locate the least loaded host, or a random idle host from a pool of idle hosts, but it is not significant how the information from which the decision is made is obtained.

All of the initial placement algorithms implemented in the experimental part of this thesis are sender initiated, and are described in section 8.1. It is therefore redundant to include an example of a sender initiated policy in this section.

Receiver Initiated

Receiver initiated load distribution [LM82, LO86] occurs when a host decides to participate in load distribution, perhaps due to finding itself below a minimum load threshold. In this case, the destination host is fixed as the initiating host, and the location policy is used to find a source host from which load may be transferred. This is a rare form of load distribution, of which early examples are the broadcast idle (BID) and poll when idle (PID) algorithms from Livny and Melman [LM82].

For the BID algorithm, when a host becomes idle, it broadcasts a status message. All hosts with more than one process wait a time inversely proportional to the number of processes on that host. The first host to broadcast a reply is the host with the most processes, and therefore obtains the idle host.

The PID algorithm avoids the broadcast facility by polling a randomly selected subset of hosts. The hosts are polled in order of their selection, and the polling host will either receive a reply consisting of a job or a negative response. If no busy hosts were in the subset, the idle host gives up.

In these two examples, the location selection is completely dependent on the communication mechanism, but the actual location selection policy of the BID algorithm is to select the most loaded host, and that of the PID algorithm is to select a random busy host.

Symmetrically Initiated

Eager et al. [ELZ86a] compared sender and receiver initiated location policies, and concluded that sender initiated policies are superior during light to medium system loads, and receiver initiated policies are more effective at high system loads. Thus the sender and receiver initiated policies can be combined to take advantage of the different characteristics exhibited at high and low loads.

The Periodic Symmetrically-Initiated (PSI) Algorithm from Benmohammed-Mahieddine and Dew [BMD94] operates in both receiver and sender modes. If the host is above threshold T, sender mode is initiated and a single request is sent to a random host. If an acceptance message is returned, the sender transfers a job to the replying host. If the host is below threshold T, receiver mode is initiated and a single request is sent to a random host. If the contacted host is overloaded, it will return a job to the receiver. If there was no reply to the original request (equivalent to a negative response) for either sender or receiver modes, the algorithm simply gives up and waits until the next period.

A similar threshold approach is from Krueger and Shivaratri [KS94], except that rather than random polling, potential senders and receivers consult lists of hosts that have identified themselves as busy or idle. The sender or receiver lists are used to poll candidates for status confirmation, until either a partner is found or a probe limit is reached.

Other Considerations

There are other considerations that the location policy may have to contend with. A good load metric is required to enable a suitable host to be selected from a set of eligible destination hosts, see section 2.3.5. The system may also be heterogeneous - in performance or configuration, in which case a multiplicity of concerns can intrude in the simple selection of a destination. Hosts of different power need to be ranked (preferably by a load metric that accounts for this) and if migration is possible between 'binary incompatible' hosts as in Theimer and Hayes [TH91] and Bishop et al. [BVW95], then this cost must be weighed against the loads on the hosts that are compatible.

2.3.3 Candidate Selection Policy

There are three main approaches to selecting candidates for load distribution:

- Select any candidate.
- Select a reasonable candidate.
- Select a good candidate.

Some systems require a process or object itself to initiate load distribution and in this case, there is no need for a global candidate selection policy. However, if a host condition triggered migration, then a candidate needs to be selected.

The following sections assume that all candidates are eligible for remote execution. In reality there are jobs that are not able to execute remotely, such as those that must access frame buffers or local filesystems. Consider these jobs excluded from the following discussion.

Selecting any Candidate

In this policy there is no attempt to filter jobs, and all jobs are considered eligible for load distribution. Examples are the random and least loaded initial placement policies, detailed in sections 8.1.1 and 8.1.2 respectively.

Selecting a Reasonable Candidate

It is clear that the previous method of selection may choose jobs or processes that are characteristically unsuitable for distribution [JXY95]. The most obvious subset of unsuitable jobs and processes are those which do not execute for a sufficient length of time to repay the cost of moving them.

Two ways in which these jobs and processes can be rejected as distribution candidates are:

- **Statically:** Processes can be analysed from system traces, and a static set of suitable processes can be used to filter candidates. This method is used by Zhou [Zho87], and Holzer[Hol96].
- **Dynamically:** A runtime record of process behaviour can be maintained, and those processes that have responded badly to remote execution can be avoided in future. This system is used in an automatic load balancer added to Sprite by Osser [Oss92].

Selecting a Good Candidate

The previous approach avoided distributing jobs and processes that are poor candidates, however, the criterion can be tightened further and the selection of a good candidate made. Initial placement and process migration, detailed in section 2.3.4, impose different restrictions in the selection of a candidate. Choosing a good candidate may require a great deal of knowledge about the job or process as well as the potential destination hosts. Indeed, it is possible that the location policy is secondary to the candidate policy, as in the Emerald system where Lehrmann [Leh93] moves heavily communicating objects so they reside on the same host.

Migration systems can obtain information about current process behaviour by monitoring each process as it executes, and good candidates can be selected by referring to this behaviour.

Initial placement is restricted to placing the job *before* execution, and as a consequence is not privy to the future activities of the job. In this case, good candidates can be selected using prediction or classification. This approach will be examined in section 3.1.

Barak and Shiloh [BS85] implemented a set of criteria to select good migration candidates in MOS. Each host periodically runs a daemon that asks processes, in a round robin fashion, if they wish to migrate. The process itself then evaluates whether:

- A lower expected response time is available on other hosts.
- More than half of its communication is with processes on another host.
- There are sufficient resources for forking on the current host.
- There is a 'favoured' host.

If these factors are favourable, the process agrees to be migrated.

Self Selection

There are systems where a process or object itself may initiate load distribution. The MOS selection scheme described in the previous section is one such system, as it is the process that decides if it wishes to migrate. The only direction from the host is to dictate when the process may consider itself as a candidate, to prevent it from spending too much time considering migration.

Another form of self selection is that used in the original Sprite system. In this system there was no automatic load distribution, and all load distribution was initiated by user level applications. One example is *pmake*, which uses as many idle workstations as it requires for a parallel version of make.

Both of these approaches fix the candidate (or forked child) and consequently fix the source host. The Chorus Object Oriented Layer (COOL) is a system that emulates a distributed shared address space over a set of loosely coupled hosts, see Lea et al. [LJP93] and Amaral et al. [AJJ+92]. Objects are the logical unit of distribution, and may move between local address spaces. If an object references a remote object, the system has two options:

- To migrate the referenced object into the physical local address space.
- To migrate the current object (activity) to the physical remote address space.

If the first option is not possible, as when referencing an object representing a system resource, then the second option of migrating the referencing object is taken. In both of these cases, the source and destination hosts are fixed, and the candidate depends on the option.

Candidate Fairness

In addition to the candidate selection criteria outlined above, there is also a 'fairness' principle incorporated into some candidate selection policies. Essentially, there are two ways of considering any benefit offered by load distribution:

- For the good of the migrated process. (greedy)
- For the good of the remaining processes. (altruistic)

Greedy implies altruistic, but altruistic need not imply greedy. Namely, the migrated process may increase its response time due to the overhead involved in moving it, and never reclaim the loss, but the processes on the host from which the process was moved will benefit, and reduce their response time. The best situation is that both greedy and altruistic occur, but if they don't, then at the very least a load distribution scheme should provide altruistic.

Greedy is often embodied as a 'fairness' principle as in [HBD95], where a candidate is only eligible if it can expect better service on the new host. However, load distribution may still be worthwhile on a global scale, even if the moved processes must be purely altruistic.

2.3.4 Transfer Mechanism

The transfer mechanism physically moves the processes from source to destination host. There has been a great deal of research in this area, including Artsy and Finkel [AF89], Powell and Miller [PM83], Douglis [Dou90] and Theimer [TLC85], concentrating on issues of transparency, residual dependencies, performance and complexity. These issues are rooted in the design and implementation of the mechanism and are outside the scope of this thesis, as are the arguments supporting the different approaches. The significant aspect of the transfer mechanism in the terms of this thesis, is the stage at which load distribution may occur.

Initial Placement and Process Migration

Distribution of workload can occur in either of two phases: tasks can be allocated to a processor before they begin execution, or they can be moved after they have begun execution. These two phases are known as *Initial Placement* and *Process Migration* respectively and are in principle independent.

Each of the two phases offer different opportunities to the policy components of load distribution. Policies that utilise migration can monitor the runtime behaviour of processes and make decisions that may involve moving existing workload as the situation demands[10]. Migration is often associated with the high cost incurred in transferring the address space of an active process, however fewer migrations may be required to balance the load. Policies that are restricted to initial placement have the advantage of a lower transfer cost, but once a process is assigned to a host it must remain there until it completes or is terminated. In depth analyses of the issues of initial placement and process migration are available in [ELZ86b, DHB95, KL88].

[10]Satisfies dynamic process behaviour - at stage 1 a process may require X and then later Y, if X and Y are best provided for from different hosts, then the best solution may be to move the process during execution to suit the different phases of execution.

2.3.5 Load Metric

The load metric is important in a load distribution scheme, as it represents the workload being distributed. There are a wide variety of possible load metrics, a few examples are:

- CPU queue length.
- Resource utilisation.
- Communication delay.
- Host idleness.
- Object affinity.

Participation, location and even candidate selections are made on the load, and it is therefore critical that the load metric is relevant. For example, Martin [Mar94] found that the number of processes on a host is a completely irrelevant load metric as it does not provide any information about the contention for resources on that host.

Fastest Response

The V migration system discussed in Theimer et al. [TLC85], used a delightfully simple ranking metric based on the group communication facility of V. When a transfer is initiated, potential hosts are polled via a multicast request. All hosts which meet an availability criteria respond, and the first response is selected. The reasonable assumption is that the first host to respond is generally the least loaded. This 'ranking' mechanism also neatly solves the problem of load communication.

Idle Hosts

If there is a reasonable proportion of idle hosts at any time in a system, then these hosts represent an allocable resource. This approach is central to many load distribution schemes that are based on the workstation model, as the 'low power' of individual workstations, combined with a high degree of idleness makes anything other than a boolean idle-busy metric unnecessary.

The idleness criteria from Sprite [Dou90], is fairly typical. An idle host must have had less than one runnable process, averaged over the last minute, and no keyboard or mouse input for the past 30 seconds.

Object Affinity

Communication may also be a measure of load, and minimisation of this requires communicating entities to reside together on the same host. A migration policy developed by Lehrmann [Leh93] in the object based Emerald system, combined two load balancing strategies. The first transfers objects from overloaded to underloaded hosts, the second moves heavily communicating objects to the same host. The second policy clusters communicating objects, and thus through communication, certain objects express an affinity for each other.

Resource Availability

A common approach is to consider the resources available on a host, and express them in a way that enables suitable hosts to be ranked.

Most systems simply consider the CPU resource, and neglect all others, while others use a more complex combination of resources, see Bond [Bon93]. Ferrari and Zhou [FZ87] explore variations on this form of load metric, with load metrics ranging from the instantaneous CPU queue length, through to a linear combination of CPU, memory and IO queue lengths. They found that for their system and set of assumptions, that the linear combination of exponentially smoothed CPU, IO and memory queue lengths produced the best performance. In a contrary finding, Kunz [Kun91a] found that single resource queue lengths were as good as a combined metric in his study with a stochastic learning automaton. These two results confirm that the load metric is important, but also indicate that the suitability of any load metric depends on the system, workload and load distribution scheme for which it is used.

2.3.6 Load Communication

Once the load on a host has been measured, it must be communicated to whatever agents make load distribution decisions. This poses a difficult problem in most distributed systems, as there is a cost involved in both the collection and distribution of the load data. There are also problems of reliability, update delays and even locating the state, any of which can result in out of date information being used to make current distribution decisions.

A number of clever solutions have been proposed for load communication, most of which are variations on one of the following five methods.

Polling

Polling is a message directed at only one host to return its current load. If polling is performed on demand it results in the most up to date information, but at a potentially high cost. The PID algorithm from Livny and Melman [LM82], the PSI algorithm from Benmohammed-Mahieddine and Dew [BMD94], and the approach from Krueger and Shivaratri [KS94], all of which are discussed in section 2.3.2, use polling to obtain current host conditions.

Broadcast

A form of undirected communication, where all hosts exchange information by broadcasting over the network. There are a number of problems with this approach:

- It may lead to an unacceptable amount of network traffic.
- All hosts receive the load updates whether they are involved in migration or not.
- In large networks there may be a substantial propagation delay over all hosts in the network.

The primary advantage is also that all hosts on the network have access to all the broadcast information, and this can be used not only to update recorded host loads on all hosts, but also to synchronise a distributed load distribution policy. A good example of this is the BID algorithm implemented by Livny and Melman [LM82], already covered in section 2.3.2.

Group Communication (Multicast)

This is a form of broadcast constrained to the members of a group. This reduces some problems associated with the broadcast method, as only members of the migration group receive the load information, but there still may be a lot of traffic. The primary advantage of the group communication system is that identification of participating hosts is simplified, making the location of suitable sources or destinations a much simpler problem. The group communication facility of the V system is used to considerable advantage in the process migration system of Theimer et al. [TLC85].

Centralised Collection Agent

Four load communication alternatives were described by Zhou [Zho87] of which three used a central collection agent:

- **Global:** All hosts periodically transmit their loads to a central collection agent. If there is no change in a host's load from the last period, then no update is sent to the collection agent. The accumulated load vector is then periodically broadcast from the central collection agent to all hosts in the system.

- **Central:** This system also has a central agent, except that the load is not periodically collected or distributed. Instead if a host wishes to offload a job, then it sends a request as well as its load to the central agent. The central agent then replies with a suggested destination. The load information held by the central agent is only updated by the distribution requests.

- **Centex:** Another approach is a combination of the Global and Central algorithms where a central agent collects all of the periodic load updates, but does not automatically distribute them. If a source host needs to locate a destination, then it contacts the central agent for the most recent information. This approach was also used by Bond [Bon93] in his Distributed Resource Measurement Service (DRUMS) and by Holzer[Hol96].

The Worm

Barak and Shiloh [BS85] use a worm in a rather creative solution to the distribution of load information between hosts in the MOS system. The idea is, that providing correct load information about all host loads is an unacceptable expense. In particular, there is a conflict between the availability of load information and the overhead required to obtain it. Thus Barak and Shiloh introduce the worm, which periodically sends half of its information vector L, of arbitrary length n, to a random host i.

Send : at each period,

$$\textit{Update own load value } L_0.$$

$$\textit{Send vector } < L_0, \ldots, L_{\frac{n}{2}} > \textit{ to random processor } i.$$

Receive : on host i and merge into own load vector by:

$$Li_h \rightarrow Li_{2h}, \qquad 1 \leq h \leq \frac{n}{2} - 1$$

$$\textit{and}$$

$$Lr_h \rightarrow Li_{2h+1}, \qquad 0 \leq h \leq \frac{n}{2} - 1$$

where Li is the local load vector on host i and Lr is the received load vector. This merge ensures only the most recent information is held, by interleaving the local and received vectors. The order of a load in the load vector implies the age of the information, and dropping half the load vector when sharing the load vector is a form of aging. There is no mechanism to remove repeated hosts from the load metric during the merge, and this may reduce the pool of possible distribution partners. Repeated elements also provide conflicting information for which there is no way of determining absolutely, which load is the more recent.

2.4 Summary

The chapter has emphasised a taxonomy for describing, comparing, designing and summarising load distribution policies. Previous work has been used to highlight its application, as well as to provide an overview of load distribution in general.

Chapter 3

Hypotheses: Balanced Resource Allocation

Chapter 2 introduced the general area of load distribution and provided a framework in which to discuss load distribution schemes. This chapter is intended to clearly identify the area of work embodied in this thesis, by firstly arguing for, and then presenting the central *process mix* and associated hypotheses.

3.1 Resource Management

All load distribution policies consider a subset of system resources as a basis for load distribution. There are two main approaches that depend on the configuration of the physical distributed system:

- Systems which are homogeneous in both capability and compatibility tend to concentrate on a single resource that they consider typifies the contention for resources [LO86, Kun91b, Oss92, GDI93, HBD95]. Systems that are homogeneous in compatibility but heterogeneous in capability may still use only one resource to represent resource contention, but scale the value appropriately.

- Purely heterogeneous systems that differ in both capability and compatibility [Bon93, KRK94, CCG95], tend to consider the capability of a host to provide the range of resources required by a process or job and distribute the workload appropriately.

Leland and Ott [LO86] have shown that processes usually fall into four categories: CPU bound, IO bound, ordinary (both unbound) and the extremely rare[1] CPU and IO bound. Devarakonda and Iyer [DI89] confirmed these categories, and observed an additional heavy memory use category.

These results imply that not only should the requirements of a process be matched to host capabilities in a heterogeneous system, but homogeneous systems may also benefit by controlling the *mix* of CPU, memory and IO bound process types on any one host. In particular, the processes already executing on an individual host combine to produce a set of remaining resources that are not homogeneous over all hosts in the system. So while each host has the same initial capabilities, the remaining capacity is different. As a simple example, consider the processes and their resource requirements summarised in table 3.1. Processes 1 and 2 are CPU bound, while processes 3 and 4 are IO[2] bound. These four jobs are to be distributed between three completely homogeneous hosts, A, B and C. A conventional CPU oriented distribution system will allocate one CPU bound process to each of A and B and the two IO bound processes to C. An obviously better arrangement is to place process 3 on the same host as process 1.

Process	CPU utilisation	IO utilisation
1	85	0
2	90	6
3	5	45
4	15	70

Table 3.1: *Two CPU and two IO bound processes. Care needs to be taken when scheduling these four processes to three hosts.*

Thus it is not only bottlenecking of the CPU that is a potential source of system degradation, but bottlenecking of IO as well. This argument can be extended to other system resources that are in demand, such as the memory system.

[1] Between 8 and 10 processes out of 8.5 million are both CPU and IO bound.

[2] Ignoring for now issues of file locality.

3.2 Hypotheses

The previous section argues that the management of all primary resources is important to the performance of any distributed system, therefore I suggest the following hypothesis:

Hypothesis 1 (Process Mix) *The distribution of a balanced mix of CPU, memory and IO bound processes to each host will lead to the more effective use of resources and delay the onset of bottlenecking.*

This process mix approach to load distribution is a logical extension of the traditional load distribution philosophy noted in Casavant and Kuhl [CK88], "...being fair to the hardware resources of the system is good for the users of that system".

This hypothesis has two major implications:

- If the system's resources are used more effectively, then individual processes will progress through the system at a higher rate.
- A means for distributing the processes within the system is required. If the resource use of a process is only available from its current execution, as in [JXY95], then the process can be migrated to a host to improve the process mix on both the source and destination hosts[3]. If *a priori* knowledge about jobs is available, then this can be used in conjunction with initial placement to distribute jobs to hosts, and so achieve a desirable mix of processes on each host.

A priori knowledge about a job for initial placement can be provided by estimating its future behaviour from its historical behaviour. Examples of prediction systems providing *a priori* knowledge for initial placement are Bond [Bon93] with a heterogeneous system, and Goswami et al. [GDI93] in a homogeneous system.

No prediction mechanism can expect 100% accuracy with a dynamic workload, and therefore the question arises of how sensitive the performance of a policy is, when relying on such predicted values.

[3] This also solves the situation where processes are nonhomogeneous in behaviour (i.e., those whose resource requirements change over its execution).

Hypothesis 2 (Prediction Accuracy) *Perfect prediction accuracy is not required nor realistic, and a reasonable estimate is sufficient for good performance.*

As prediction can provide *a priori* knowledge to load distribution policies, it should be possible to place a job correctly, and therefore it will be unnecessary to later migrate that job after it has begun execution. A job that is correctly placed is one that behaves as predicted and results in resource demands that are consistent with the policy by which it was placed.

Hypothesis 3 (Initial Placement) *Initial placement can use prediction to attain performance comparable to that of process migration.*

Workloads containing interactive processes suffer a degree of randomness and therefore predictions are not always accurate. In particular, any system that relies totally on prediction is likely to suffer from this degree of randomness that can not be accounted for by the load distribution policies. Therefore hosts should still be monitored to ensure they are consistent with their computed loads.

If the initial placement policies using prediction are indeed sensitive to this inaccuracy, hypothesis 3 may lack support. One way around this, is to accept that mistakes are made with initial placement, and use another mechanism such as migration to provide a degree of correction. Many systems, such as Sprite [DO91] and V [TH91] provide both initial placement and process migration systems, yet do not integrate the use of both mechanisms in a single load distribution policy.

Hypothesis 4 (Combination) *Combining initial placement and process migration in a single load distribution policy, will enable placement errors to be corrected, thus improving system performance.*

3.3 Summary

This chapter presents an argument that the composition of the workload on an individual host is important, and suggests that by maintaining an appropriate mix of processes based on the resources that they consume, the use of system resources will be improved and the onset of bottlenecks reduced. This argument is embodied in the first and central *process mix* hypothesis.

There are two physical means for distributing processes to adjust the mix of processes. Migration can monitor the activity of a process and make adjustments to the workload composition at any time. Initial placement is more restrictive, as distribution occurs before any activity can be monitored. Thus policies for initial placement mechanisms require more information than is available at distribution time and must use some form of prediction. Thus the *prediction accuracy* and *initial placement* hypotheses are presented while assuming the *process mix* hypothesis.

The final *combination* hypothesis suggests that if the *initial placement* hypothesis fails, the combination of initial placement and process migration may solve the prediction accuracy problem.

Chapter 8 describes the algorithms developed to test these hypotheses and chapter 4 details related work.

Chapter 4

Related Work

Chapter 3 presented the hypotheses that form the basis for the thesis. The related work detailed in this chapter is mostly concerned with providing or using resource or lifetime information to make placement decisions.

Providing this information falls into two broad categories:

- Probabilistic models are derived from analysing workload data, and determining relationships from which a probability function can be derived. These models can then be used directly as in section 4.1.2 or indirectly as in section 4.1.1.
- Historical data from previous presentations of a process may be used to predict its future behaviour providing a form of *a priori* knowledge. The advantage of using historical data is that a prediction can be made *before* a process is started, and is therefore suitable for initial placement. There are two ways in which historical data can be used for load distribution:
  - **Explicitly:** Given a job, a prediction can be made about its behaviour (perhaps it's lifetime or resource use), and this information is used to distribute the job conventionally (i.e., the prediction and distribution systems are distinct). Examples of this approach are discussed in section 4.2.
  - **Implicitly:** Given both a job and the system, a learning mechanism may use previous experience to recognise and schedule the job to a suitable host. In this case, the prediction (if there is one) and distribution systems are intrinsically combined. Two different implicit approaches are discussed in section 4.3.

4.1 Probabilistic Models

Probabilistic models are derived from analysing real workloads and using relationships therein to predict future behaviour. The two approaches presented are based on the same process lifetime probability distribution model in which older processes have a higher probability of remaining in the system than younger processes. The difference between the two models is the way in which they schedule the processes.

4.1.1 Process Lifetime Model

Leland and Ott [LO86] investigated the behaviour of 9.5 million UNIX processes and found that there was on average, an almost perfectly linear relationship between the current age of a process and its expected residual CPU time. This resulted in the rejection of any type of exponential distribution for process lifetimes, and the finding that it is better to distribute old rather than new processes due to a long thick tail on the actual distribution.

They consider the theoretical *Spiral Assignment* of processes to hosts by age as the basis of a load distribution policy. Essentially, if all L processes p_i in a system are sorted by age A, so that at some time t:

$$Ap_1(t) \leq Ap_2(t) \leq \cdots \leq Ap_i(t) \leq \cdots \leq Ap_{L(t)}(t)$$

Then the spiral assignment assigns the processes to hosts such that:

$$host_i \leftarrow p_i, p_{(i+N)}, p_{(i+2N)}, \ldots$$

where N is the number of hosts in the system. This arrangement has a number of advantages:

- Young processes have a low probability of remaining in the system and thus the most likely process to leave is p_1 (mapped to $host_1$). As a new process will be the youngest, it will be assigned to $host_1$. This property helps maintain the spiral arrangement.

- There is a mix of process ages on each host which Leland and Ott implicitly correlate with balanced load.

The Spiral assignment policy as given, requires instantaneous and free reshuffle of the set of processes over the set of hosts, and in this sense is not practical. Thus Leland and Ott developed an approximation to the spiral assignment based on the sum of the ages of all processes younger than process p_i on the same host ($CRIT_i$). If the oldest process p_i on a host has a $CRIT_i$ greater than a $MINCRIT$ threshold, then that process is eligible for migration to an idle host. This policy is implemented as receiver-initiated, if a host is idle, it calls for bids and the location selection policy selects the host h with the highest load that also has some processes $p_m, \ldots, p_n$ above the MINCRIT threshold. The candidate selection policy then selects the process p_i with the greatest $CRIT_i$ value from host h.

4.1.2 Process Lifetime and Movement Cost Model

The work by Harchol-Balter and Downey [HBD95] is a logical continuation from the work by Leland and Ott. In particular, the empirical observation that migration of old processes is superior to that of migrating young processes, forms the basis of the policy developed by Harchol-Balter and Downey.

The primary departure is that Harchol-Balter and Downey use the process lifetime distribution probabilities explicitly. Rather than assigning processes by age in a spiral assignment, and always migrating the oldest process, they use a two tier candidate selection approach for selecting processes to transfer from the most loaded $host_m$ to least loaded $host_l$.

- A set of n eligible processes $p_1, \ldots, p_n$ is formed from those processes on $host_m$ that would obtain 'better' service on $host_l$.
- Then the best candidate from this set is selected with the following candidate selection criterion:

$$max\left(\frac{p_1\ time\ alive}{p_1\ cost\ of\ migration}, \frac{p_2\ time\ alive}{p_2\ cost\ of\ migration}, \ldots, \frac{p_n\ time\ alive}{p_n\ cost\ of\ migration}\right)$$

This approach is described in more detail in section 8.2, as it is one of the algorithms used in the experimental part of the thesis.

4.2 Explicit Load Distribution

The following sections all detail different approaches to using some form of prediction to improve the quality of load distribution. Workload classification is a common research topic [CF86, BB94, Raa93], however there has been less research on how it can be applied to load distribution, two examples are discussed in sections 4.2.2 and 4.2.3.

4.2.1 Average

Prediction need not be complex, and in some situations, quite simple schemes can prove useful. If all processes in a system are homogeneous, or at least similar, then a static constant can be used to estimate the resource requirements. The simplest dynamic solution is persistence, where the resources required by a process during the most recent execution are predicted for the next execution. A running average is another simple approach for providing predictions, and can be combined with persistence to favour the values from more recent executions.

The use of averages is a common technique, examples include Bond [Bon93] who used averages of CPU, IO and memory in a prototype of the Simple Task Allocation using Resource Selection (STARS) system, and Osser [Oss92] who used average lifetime to exclude short jobs from remote execution.

4.2.2 Load Distribution without Host State

Goswami et al. [GDI93] used the state transition prediction mechanism developed by Devarakonda and Iyer [DI89] to provide estimates of process resource use. This estimate is then used to compute the host loads for placement.

State-Transition Model

The prediction system uses states and state transition probabilities to estimate the future resource requirements of a process.

A k-means[1] clustering algorithm was used to identify seven high density clusters in a 3 dimensional space (CPU, memory and IO). The centroid of each cluster was defined as a

[1] *minimise* $\sum_{i=1}^{k} \sum_{j} (x_{ij} - \overline{x})^2$, where $x_{ij} \in C_i$, C_i is the i^{th} cluster of k and $\overline{x}$ is the centroid of C_i.

state representing process resource usage. The states in general agree with the categories of process found in Leland and Ott [LO86], with CPU bound, IO bound and neutral categories. In particular, an approximate total of 20% of all process instances were found to fall into the CPU bound and IO bound categories (roughly 10% each) and an additional category of heavy memory use was identified that consisted of around $2\frac{1}{2}$% of all instances. The processes falling into the remaining four categories were all light resource consumers.

With the observation that a process may change the state it occupies between separate executions, transition probabilities are defined between states for each program, and the last N program executions are used to estimate the state transition probabilities tPr:

$$tPr_{ij} = \frac{transitions\ from\ state\ i\ to\ j}{number\ of\ transitions\ from\ state\ i}$$

Prediction Scheme

Given the state-transition model for a program, its future resource usage is predicted by:

- The past N executions of p are sorted into sets belonging to each of the seven clusters (states).
- Each of these sets forms a subcluster, and the centroid of this subcluster is taken as the program's resource usage in this state.

The program's resource requirements r, are computed by multiplying each of the transition probabilities tPr_{lj}, from the state occupied during the process's last execution l, with the centroid of each subcluster d_j:

$$r_k = \sum_{j=1}^{N} tPr_{lj} d_{jk}, \qquad k = CPU,\ IO\ or\ memory$$

where N is the number of clusters (seven).

Load Metric and Communication

Goswami et al. argue that conventional load distribution systems require too great an overhead if the load information is communicated frequently, and note that performance of the load distribution falls as the frequency of load updates is reduced. They propose that proper initial placement of processes based on predicted resource requirements will produce better response times with lower communication overheads.

The essential idea is that the load on a host need not be collected, but may be computed from the resource requirements of scheduled processes. This unconventional approach means that the only updates required by the load distribution mechanism are notification of process termination.

The scheduler now takes on the responsibility of providing the load on each host, without measuring the state of the host. This is achieved by treating predictions as 'perfect' and recording the estimated resource usage from each placement made to a host. These values are then used to compute the 'load' on each host before another placement is made:

$$load_i = \sum_{p=1}^{N_i} \frac{CPU\ required_p}{CPU\ required_p + IO\ required_p} \tag{4.1}$$

where

- $load_i$ is the load on $host_i$
- N_i is the number of processes on $host_i$
- $CPU\ required_p$ and $IO\ required_p$ are the predicted CPU and IO requirements for process p.

The load is updated when a process starts or finishes. When a process terminates it is buffered, and the updates are periodically sent to the scheduler. The scheduler removes the terminated processes from the host load computation and then passes the termination information to the prediction system for further learning.

Policies

Goswami et al. developed four load distribution policies, two centralised and two distributed. The centralised policies are sufficient to explain the approach, and as discussion of the distributed policies will not contribute to the explanation, they are omitted.

The first placement policy was a simple scheme that used the computed CPU load to select the least loaded host. The second policy estimates the expected response time rt_i that a *job j* will receive at each *host i*, and then selects the most favourable:

$$rt_j = \sum_{p=1}^{N_i} min(CPU\ required_p, CPU\ required_j)$$

where N_i is the number of processes on *host i*.

This equation sums the CPU requirements of all processes on $host_i$ that are smaller than the new process, and for each process larger, the new process's CPU requirement is summed. The estimates the response time experienced with a round robin scheduling algorithm.

This is primarily to avoid problems with round robin scheduling and too long a quantum.

Summary

Goswami et al. use a prediction system for providing a form of *a priori* information about processes before execution. The information is used to compute the load on the hosts in the system, rather than obtaining this information from the hosts themselves. No attempt is made to perform any kind of candidate selection, other than a simple age filter in one of the distributed algorithms.

Due to the CPU load computation from equation 4.1, the load estimates will be exaggerated for interactive processes or processes waiting on timed events. This could reduce the load distribution to random, as no checking is performed to ensure that the load estimates are representative of the actual system load. The load computation method also requires that the load distribution is pervasive, as load that is not under direct scheduler control will invalidate the estimates.

4.2.3 Weighted Resource Allocation

Bond [Bon93] also employs classification in STARS to predict the requirements of a job prior to execution, but exploits this information to select the best site for executing the process in a system of heterogeneous[2] hosts.

[2]Heterogeneous in both capability and compatibility.

Prediction Scheme

The prediction scheme developed for STARS, classifies tasks incrementally and seeks to overcome shortcomings evident in the prediction mechanism from Devarakonda and Iyer [DI89]. In particular, Bond claims that additions to the state transition model require regeneration of the entire model, and this alone makes it unsuitable for incremental updating. In addition, Bond suggests that prediction can be improved by considering additional predictors, such as group and time rather than simply relying on program name.

Load Distribution

STARS begins by forming a set of compatible hosts for executing a new job. This set of eligible hosts is then ranked by capability, using a set of weights that reflect the relative importance of resources to the particular job. A typical weight vector for a CPU bound program may consist of:

$$\textit{max cpu idle}\ 0.25,\ \textit{max mips}\ 0.25,\ \textit{min load}\ 0.5$$

which gives $\frac{1}{4}$ weighting to an idle CPU, $\frac{1}{4}$ weighting to a powerful processor and $\frac{1}{2}$ weighting to a minimum load. These weights are then applied to the resource availability of each eligible host, and the highest value indicates the most suitable host for execution of the job.

STARS and the prediction system were not completely integrated, and the system tested in Bond's thesis used moving averages to tailor the resource weights for each job.

4.2.4 Tracing

In a novel approach, Ju et al. [JXY95] trace a process through the first second of its life, to identify processes suitable for remote execution. If the process is suitable, being non interactive and non IO bound, then the process is terminated and restarted on an appropriate host.

Predicting Future Behaviour

Processes are only traced for one second from the start of execution. Therefore Ju et al. require the assumption that a process will exhibit the same behaviour over its entire lifetime as it did in that first second.

The data collected during the trace period is intended to determine if the process is suitable for remote execution. Ju et al. collect the total memory requirement, the length of all opened files, the total CPU time and the amount of data read and written to memory, and to files. These values are used to estimate the CPU and IO utilisation of the process, and the speed of memory and file access.

The expected lifetime τ, of a process, is estimated from the time required to access *all* memory pages allocated to the process and to read all opened files:

$$\tau = (allocated\ pages \times access\ time) + (open\ files \times access\ time) \tag{4.2}$$

If the program has been executed before, then the values obtained during previous executions are available to adjust the lifetime estimate from equation 4.2.

Load Distribution

An eligible candidate is preferably CPU bound, has an expected lifetime τ of at least ten seconds and is non interactive.

The process is considered interactive if any of the open files are stdin[3] or stdout. The recorded CPU and IO utilisation is used to classify the process as CPU ($> 70\%$ utilisation) bound, IO ($> 30\%$ utilisation) bound or neutral.

If the candidate is eligible, it is terminated and restarted on the least loaded host.

4.2.5 Best Fit Resource Allocation

Cena et al. [CCG95] take *a priori* information as given, and concentrate on the actual distribution of load to hosts. Although a neural network is used, load distribution is explicit, as the *a priori* information is provided to rather than by the neural network.

[3]Stdin and stdout are UNIX files intended for writing to and from the console.

Load Distribution via Best Fit

This location selection policy is designed for a heterogeneous[4] distributed system, and as a consequence, focuses on the capability of a particular host to supply resources required by a job.

The prototype system divides available system resources into available memory and available CPU (normalised). Processes are characterised by the required amount of memory and the desired response time. A conventional best fit algorithm, as described in section 8.3.2, is used to allocate processes to the available resources in the following order:

1. Determine the set of hosts that can satisfy the memory request.
2. Select the host that leaves the minimum unallocated CPU while still satisfying the request.

If the memory request can not be satisfied, then the process can not be executed. If however there is insufficient CPU, the request is satisfied with the best available (as the response time is desired, not mandatory).

Neural Network Implementation

There is no requirement for the fitting to be implemented by a neural network, indeed, the training set was computed conventionally. However a stated goal of LAHNOS (Local Area Heterogeneous Operating System) for which the scheduler was designed, is to increase the 'automation' of the system and therefore a brief description of the implementation is included for completeness.

The distribution algorithm was implemented by training a feedforward neural network using the Back Propagation (BP) learning rule [HN90].

The input vector encodes the requirements of the job and the current availability of the system (restricted to three machines in the prototype system). The output is a single value representing the selected host.

[4] Heterogeneous in capability rather than compatibility

4.3 Implicit Load Distribution via Learning Mechanisms

Learning mechanisms are used to make placement decisions, in which, depending on the success of the last placement, the decision is reinforced or degraded. The candidate and location selection policies are combined into a single operation, and the policies are therefore implicit.

4.3.1 Stochastic Learning Automaton

A basic stochastic learning automaton is a set of permissible actions[5] (i.e., place a job on machine X), each of which has an associated probability. The probabilities associated with each action are modified depending on the success of each placement decision (the learning rule). If a poor decision is made, the probability associated with that action is decreased (penalised) and if the decision is good, then the probability is increased (rewarded).

Kunz [Kun91b] extends this basic approach by giving the automaton a set of states, each of which contains a complete set of permissible actions and associated probabilities. A mapping derived from the system status (load) is used to select the appropriate state from which action probabilities are drawn.

Figure 4.1 illustrates the automaton from host four of a five host system. The set of permissible actions in each state do not include self assignment to host four.

The mapping in Kunz is based on the current status (load conditions) of the system as a whole. The use of multiple states increases the stability of the stochastic learning automaton as it allows different load conditions to be represented. State 1 in figure 4.1 is as yet unvisited and shows the initial condition of a state - uniform random, the other three states demonstrate varying degrees of learning.

Learning

The probabilities of the stochastic learning automaton are adjusted depending on the success of a job placement. An error function is a means of determining the success of a

[5]Self assignment is not a permissible action, and therefore the set of actions does not include an assignment to the local host.

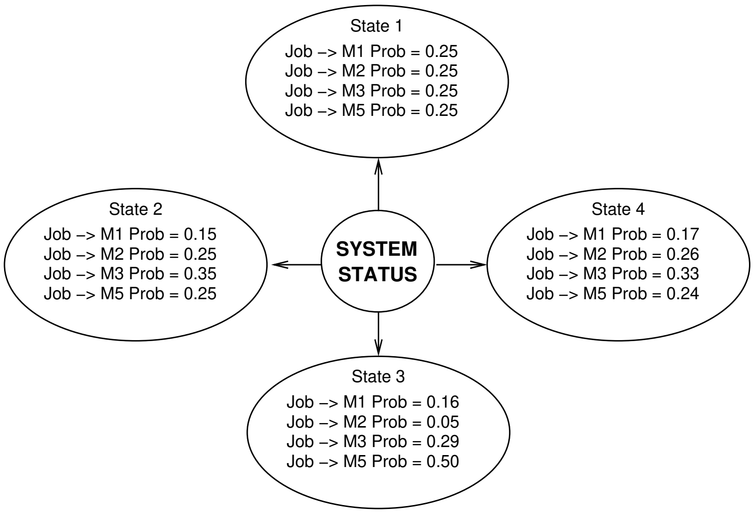


Figure 4.1: *A single stochastic learning automaton.*

placement, and a learning rule is used to adjust the weights. Kunz investigated three error functions and three learning rules for training the automaton. The most basic error function is a binary measure of goodness δ, where a placement is considered 'good' if the destination host was underloaded, and 'bad' if the destination host was overloaded. The other two error functions return quantitative rather than qualitative results:

$$\delta = threshold - current\ load$$

$$and$$

$$\delta = cs.load - rs.load$$

where cs[6] is the source and rs is the destination host.

It is obvious that the two quantitative error functions require different learning rules to control the learning with the extra error information, however the binary case is simpler and provides a suitable example for demonstrating this approach.

The basic learning rule is the linear reward-penalty scheme, where the following reward and penalty functions are used to modify the current action probabilities:

[6]These names are the compute server and remote server. They are in this form to maintain consistency with the conventions adopted in chapter 8.

$$reward[Pr_j(n)] = A \times Pr_j(n)$$

$$and$$

$$penalty[Pr_j(n)] = \frac{B}{(r-2)} - B \times Pr_j(n)$$

where

- $0 < A < 1$ and $0 < B < 1$ are reward and penalty constants respectively,
- r is the number of hosts in the system,
- Pr_j is the probability of choosing action j at time n.

The reward and penalty constants A and B are used to control the speed of probability adjustment (learning).

Kunz's learning automaton does not consider the job being scheduled. This is not a restriction of the stochastic learning automaton itself, as the job could be used as a factor in the mapping of states.

A later study by Schaerf et al. [SST95] into multiagent stochastic learning for load distribution is also of interest, and further extends the approach outlined above.

4.3.2 Adaptive Linear Element

Koch et al. [KRK94] developed a load distribution scheme based on a single *ada*ptive *line*ar element (Adaline[7]), illustrated in figure 4.2. The Adaline computes the weighted sum of an input vector **x** with a weight vector **w**:

$$y = \sum_{i=0}^{n} x_i w_i$$

resulting in an output y.

The input vector **x**, is comprised of elements representing the current load on the system. The job and machine are not represented in the input vector, but are instead used to select the weight vector **w** used by the Adaline to compute the weighted sum y. Each job and machine pair is represented by it's own weight vector, a strategy similar to Kunz's additional states, to improve the stability of the Adaline.

[7]Developed by Widrow and Hoff, 1960.

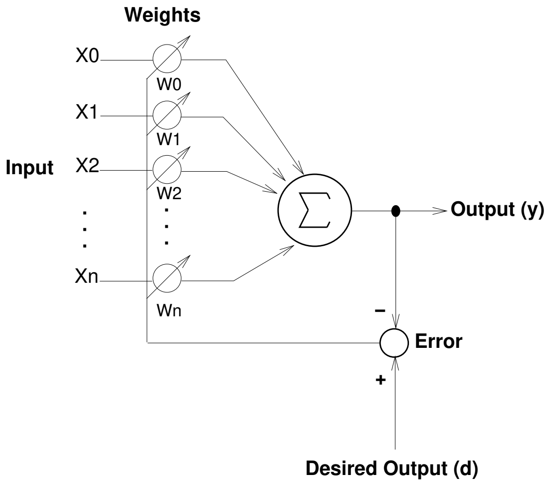


Figure 4.2: *The adaptive linear element or Adaline.*

Load distribution

On presentation of a job j, the Adaline is used to predict the expected execution time y for j on each machine. The machine with the shortest expected response time for j is selected and j is initiated there.

Learning

The Adaline is trained by measuring the difference δ between the expected execution time y, and the actual execution time d:

$$\delta = d - y$$

The δ term (error) is then used to update the weights using the standard Widrow and Hoff Least Mean Square (LMS) delta rule:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_k\mathbf{x}_k$$

where

- $\mathbf{w}_k$ is the weight vector associated with the job and the host on which it was executed,
- $\mathbf{w}_{k+1}$ is the result of learning from the placement of the job,
- $\mathbf{x}_k$ is the input vector,
- α is the learning rate (i.e., the size of each weight adjustment).

4.4 Summary

The systems presented in this chapter have one factor in common, all have attempted to make better load distribution decisions by using more information about resources, or relationships between jobs and hosts.

The two probabilistic models use the current age of a process to predict its expected lifetime. The remaining seven systems use learning mechanisms, as simple as averaging or as complex as hierarchical clustering, to either provide a form of *a priori* information about a program (explicit), or to perform the entire load distribution from experience (implicit).

None of these previous investigations do more than superficially overlap the area of investigation, and thus none provide an alternative approach with which to compare the results or progress of this work.

Part II

Chapter 5

A Testbed for Load Distribution Policies

Load distribution is a difficult system service to evaluate, its actions may be observed, but its performance can only be assessed by measuring its *side-effects*. Side-effects commonly used to gauge the performance of load distribution include metrics such as: average process response time; utilisation rates for cpu, disk and the network; and mean CPU waiting time. Thus the evaluation and comparison of load distribution policies requires a *testbed*, on which these and other side-effects can be measured.

5.1 The Testbed Design

The primary requirement of the testbed design is that it needs to facilitate the testing of the hypotheses introduced in section 3.1.

There are three major components in the testbed:

- The distributed system model.
- The load distribution mechanism.
- The load distribution policy.

Figure 5.1 illustrates the conceptual design of the testbed, with the load distribution policy determined by the current experiment. The centralised load distribution policy shown is not essential, but is better for illustrating the separation between the components.

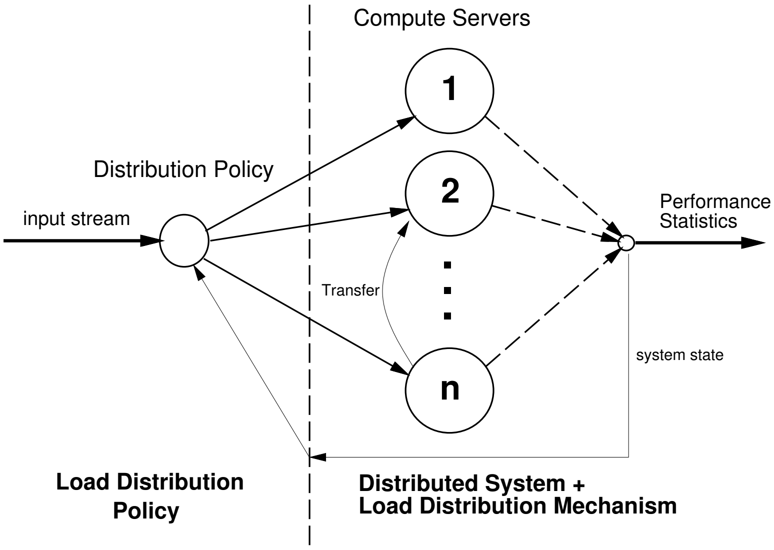


Figure 5.1: *A testbed for comparing load balancing strategies on a distributed system with n compute servers.*

With this testbed, load distribution policies can be tested and compared using the same underlying distributed system and load distribution mechanism. Section 5.2 argues the choice of a trace driven simulation for the distributed system part of the testbed, and chapter 6 describes the actual simulator. Chapter 7 details the load distribution mechanism, and chapter 8 presents the load distribution policies and their algorithms.

5.2 Methods of Analysis

There are a number of different approaches to representing the distributed system component of the testbed. The three following approaches outline the main alternatives and argue their applicability for this study. All three have their own advantages, consequently there is no completely correct choice.

5.2.1 Theoretical Models

Theoretical models are excellent for analysing simple systems, and models already exist for static load balancing systems. Payne [Pay82] states that, "a system that can be analysed mathematically, will generally provide more accurate results and more information with potentially less effort than simulation". However the more complex and interesting dynamic models tend to be intractable [Pay82, WM85].

Systems involving nonlinear elements such as queues and random influences can often be analysed with queueing theory. Unfortunately these solutions are only available for limited dynamic cases, and when different algorithms and distribution architectures are being studied, a change in the parameters of the system often requires a new model.

The system to be modelled is an unrestricted distributed system: potentially heterogeneous in composition, containing resource queues, and fundamentally dynamic in nature. Any one of these make the system a poor candidate for mathematical modelling.

5.2.2 Direct Measurement of a Real System

Experimenting on a physical system has the primary advantage of realism. There are no simplifying assumptions that could invalidate the results, other than in the workload (when synthetically generated), or in the measurement of the effect on the system. There are however, some properties in a real system that reduce its desirability.

- Measurement hardware is specialised, inflexible and expensive.
- Interference can be introduced into the system by software measurement.
- A real system is too complex to account for, or control every clock tick and interrupt. As a consequence experiments lack repeatability and even an entirely synthetic load will incur variation across runs.

These problems are minor when compared to the value of the results from a real system, however, the experiments required for this thesis need to be repeatable. Otherwise valid comparisons and therefore support for the hypotheses can be difficult to make. Additionally the cost in implementation time as well as experimental time is too great.

5.2.3 Simulation

Simulation is a tool for system analysis that has been studied for a considerable length of time. In this respect it is a well understood discipline, with well defined areas of application. Shannon [Sha75] provides the following formal definition:

> *Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system.*

An alternative definition of simulation comes from Franta [Fra77]:

A system is a big black box
Of which we can't unlock the locks,
And all we can find out about
Is what comes in and what goes out
Perceiving input-output pairs,
Connected by parameters,
Permits us, sometimes, to relate
An input, output and a state.
If this relation's good and stable
Then predict we may be able,
But if this fails us - heaven forbid!
We'll be compelled to force the lid!
Kenneth Boulding

The last part of Shannon's definition describes perfectly the purpose of this thesis, without the costs inherent in building or modifying a real system. Simulation can also deal with more complex systems and interactions between systems than mathematical models. Other advantages are:

- Simulation is a completely controlled environment and consequently repeatable experiments can be performed.
- The software basis of simulation makes it flexible as system parameters and functions can be easily changed.
- Simulations are not restricted to real time. Experiments can be sped up or slowed down as required.
- Simulations have been used successfully in previous studies of this area, in particular, Eager et al. [ELZ86b], Benmohammed-Mahieddine and Dew [BMD94], and Zhou [Zho87] encourage its use.

The major limitation of simulation is not in the implementation, but in the validity of the models used to construct the simulation itself. Flawed models can destroy any value that the simulation results may have. For this reason alone, the process of system development suggested by Payne [Pay82], recommends that simulation should be used as a stage before the use of a prototype, to verify a hypothesis prior to the commitment of costly resources. It is this support of the hypotheses that is desired from this thesis, and therefore simulation is indeed a suitable choice for the distributed system component of the testbed.

5.3 Workload Data

There are three main approaches to providing a workload description for the simulation:

- **Stochastic** models of the workload have the advantage of being derived from probability distributions and are therefore statistically well defined. Stochastic modelling is a method often introduced for dealing with uncertainty (when there is variation but no discernible pattern), Franta [Fra77]. The main advantage of this approach is that parameters can be altered to provide widely different test situations. The simplifying assumptions used in creating a stochastic model can be difficult to defend, yet this is a popular approach.

- **Traces** of jobs started on a real system are recorded and used to drive the simulator. Provided a true cross-section of activity is recorded, this will provide the most realistic data. In addition, traces can be rerun for repeatability, and different traces can be used to study different workload situations. The drawback of this approach is that it is still a sample, and the data may only reflect the activity on the system from which it was recorded. Another problem is that obtaining detailed trace data from a real system, may interfere with the activities being recorded, thus reducing the value of the measurements.

- **Synthetic mixes** are obtained by observing the system under study, deriving rules and relationships to create a facsimile of the real situation. Like a simulator this approach has the advantage that it is under complete control and can be used to show effects on demand (i.e., rare situations can be created and the implications for the system can be studied). Care must be taken when constructing the models for a synthetic workload, as any results are open to challenge if the models do not represent a realistic workload.

All of the above methods require a description of a workload for which results can be obtained and analysed. Experiments should be repeatable and poor workload descriptions must be avoided on the principle of 'garbage in, garbage out'.

With this in mind, the *process mix* hypothesis is most effectively tested with a real workload composition, as it is important that the proportions of the resource bound processes are realistic. A synthetic or stochastic model could easily misrepresent the system load, and lead to faulty conclusions. Therefore the most reliable choice is to use system traces to drive the testbed simulator.

Chapter 6

A Distributed System Simulation

The primary aim of this chapter is to introduce and describe the simulated distributed system component of the testbed outlined in chapter 5.

6.1 Distributed System Model

Earlier simulations such as those used by [ELZ86b, LO86, Zho87, BMD94, HBD95], tended to accurately represent only the CPU resource in the system, and while adequate for those studies, evaluation of the *process mix* hypothesis requires a full set of resources to be modelled.

As load distribution is concerned with the interaction of a set of processes and compute servers[1], these two components obviously form the basis of a model for the distributed system simulation.

- **A Set of Processes:** Each process is characterised by the set of resources it requires to finish and the rate at which each resource is consumed.
- **A Set of Compute Servers:** Each compute server consists of a finite set of resources for consumption by processes.

There are two possible approaches to constructing a model from these components. Firstly the processes could be passive, and the compute servers the active component, dividing its resources evenly between its processes, paying no attention to the rates at

[1] As this chapter deals with a simulation based on a real system, the generic term 'host' is replaced with the more accurate and specific term 'compute server'.

which different process consume different resources. A more realistic approach is for the the processes to be active, driving the simulation with their processing demands, leaving the compute servers as simple providers of system resources and arbitrators of contention.

This leads into an explanation of the structure of the chapter. As processes are the active agents in the simulation, the first half of the chapter explains the modelling of the processes, and the way in which they flow between resources. The second half of the chapter details the simulation of the compute server, and its service rates.

6.2 Process Modelling

A process is a program in execution and as such consists of code, data and machine state such as the stack register. This model is more complex than is needed, and can be simplified by considering a process to be characterised by the resources it uses or visits during execution. The significant resources are:

- The amount of CPU time.
- The number of file IO transfers.
- The number of page transfers.
- The number of external events[2].
- The amount of memory used.

Section 5.3 presented the three main options for providing a workload to drive the simulator, with the conclusion that a system trace offered the most suitable data. The most accessible form of a system workload trace is the accounting log. The accounting log also has the advantage that it is collected automatically by the system, and therefore does not cause any additional interference to the system being measured.

6.2.1 Resource Accounting

The accounting log provides the following summary of a process's activities in the system:

[2] User interactions and timer events. This is discussed in section 6.2.8.

- The command name.
- The user and group identifiers.
- The terminal.
- The amount of user and system CPU time used[3].
- The time the command started and finished execution.
- The elapsed time of the command.
- The average amount of memory used.
- The total number of bytes transferred, described in 64 byte blocks[4].

The simulation also requires the following process information that is not provided by the accounting system:

- The paging activity in which the process was involved.
- How many IO blocks were local, and how many were remote.
- When IO and paging events occurred.
- The time spent waiting on external events.

The only other caveat is that the reported values are statistically gathered by the accounting software, and are therefore not exact. This is of minimal concern, as the level of simulation is quite coarse, however, in addition to this inaccuracy, section 6.3.2 finds that the reported memory values do not reflect the savings made from code sharing. I will, defer discussion on this point to section 6.3.2.

As the objective is not a complete reconstruction of previous events, but rather the representation of a realistic load, these small inaccuracies are not expected to be critical to the validity of the simulation. The lack of paging, file IO and user information is more serious, but these can be reconstructed by modelling these events from the information that *is* present in the accounting logs.

[3]CPU user time is the time spent executing user level code, while system time is the time spent executing kernel code. This user time should not be confused with the time spent interacting with a user.

[4]The only information is the number of bytes, the use of 64 byte blocks in the accounting log has nothing to do with the size or frequency of transfers.

Trace Data Collection

To provide a representative sample of system activity for the system trace, the accounting logs from five Sun IPX compute servers used by second and third year computer science undergraduates were recorded over a one month period. These five compute servers and an associated fifty X-terminals follow the Xterminal-Server model, and therefore it is this model that is used in the simulation. All five compute servers are peers, with the students' home file systems divided evenly amongst them.

In addition to the accounting log information, the size of the program file was included in the trace.

Arrival Time Resolution

The start and finish times from the accounting log have a resolution of one second. This results in multiple jobs with the same apparent starting time, and up to thirty 'simultaneous' jobs occurring during some peak periods. This is an unrealistic situation, as in reality the jobs would be distributed within the interval. To correct this, the arrival times were modified by adding a two digit random number to represent hundredths of a second. This increases the resolution of the starting times, distributing the jobs bursts over the entire one second interval.

6.2.2 Processes

To restate, processes are consumers of resources such as CPU time, and each compute server is a provider of a finite set of resources. The compute server is responsible for controlling access to its resources, however the process determines which resources it wants to consume, and in what order. This means that the process may be in any one of the following states:

- **Resource Use States:**
  - **Ready:** The process is waiting in the ready queue for the CPU
  - **Running:** The process currently holds the CPU.
  - **IO service:** The process is waiting for an IO request to complete.

  - **Paging:** The process is waiting for a page request to be filled.
  - **User:** The process is waiting on a user event to complete.

- **Control States:**
  - **Swapped:** The process has been swapped out.
  - **Migrating:** The process is in migration and is logically not in the compute server.
  - **Finished:** The process has finished execution, and the empty process is waiting to have its runtime statistics collected.

The resource states represent the process accessing or waiting to access a specific resource during its execution. The control states represent the states were the compute server dictates the actions of the process. The following sections are dedicated to explaining the entry into each of the resource states. The swapped control state will be discussed in section 6.3.3, the migrating control state is explained in section 7.3.2, and the finished control state is described in section 7.4.

6.2.3 Process Flow

The processes within a compute server move between the states defined in the last section. The actual flow of a process is determined by it's probability Pr of visiting each state, which is calculated from the remaining resource requirements of the process.

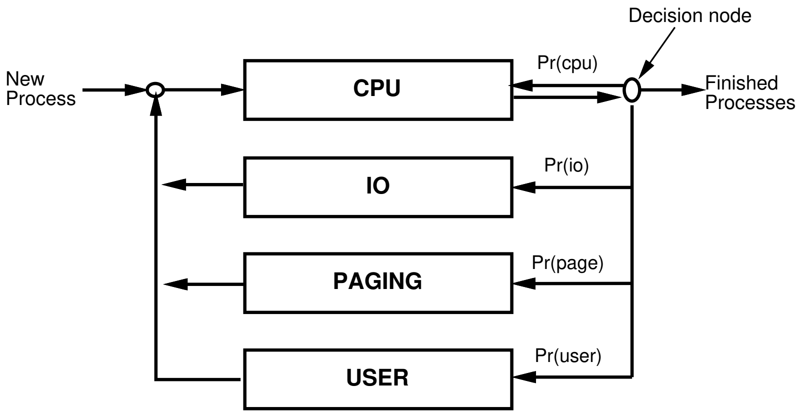


Figure 6.1: *A process's view of it's compute server is as a set of resources which are visited until all of the process's resource requirements are satisfied.*

The process must pass through the CPU state before any other state. In particular, a process may only initiate a page transfer, file IO or user event after first obtaining the CPU. A ready process enters the CPU and at the end of each CPU chunk[5], the probability Pr of the process returning to the CPU or utilising some other system resource, is computed in the decision node from the remaining resources required by the process p.

- The probability of returning to the CPU (if the quantum has not expired) is:

$$Pr_{cpu} = 1 - (Pr_{io} + Pr_{page} + Pr_{user})$$

- The probability of an IO event is:

$$Pr_{io} = \frac{p_{remaining\ io}}{p_{remaining\ cpu} + p_{remaining\ io} + p_{remaining\ pages} + p_{remaining\ user}}$$

  Where the remaining CPU visits are defined as the remaining time divided by the current chunk size, see sections 6.2.4 and 6.2.5.

- The probability of a paging event is likewise:

$$Pr_{page} = \frac{p_{remaining\ pages}}{p_{remaining\ cpu} + p_{remaining\ io} + p_{remaining\ pages} + p_{remaining\ user}}$$

  see section 6.2.6.

- As is the probability of a user event:

$$Pr_{user} = \frac{p_{remaining\ user}}{p_{remaining\ cpu} + p_{remaining\ io} + p_{remaining\ pages} + p_{remaining\ user}}$$

  see section 6.2.8.

As a process consumes its resources, the relative probabilities will change to reflect the proportion of remaining resources. So these probabilities must be recomputed every time the decision node is entered, to ensure that the correct number of visits are paid to each of the resources, preceded by a CPU visit.

Given the probabilistic flow of processes within the compute server, the rest of this section will detail the computation of the resource totals, from which the initial probabilities are computed.

[5]A CPU chunk is the defined in section 6.3.1

6.2.4 CPU Usage

The total time a process spends on the CPU is the sum of the user and system times from the accounting log. A process terminates when its CPU time is fully satisfied, regardless of any other unconsumed resources. To minimise the possibility of resources remaining at the end of a processes execution, the time that a process spends on the CPU during each visit is adjusted to allow one more visit to the CPU than there are remaining resource visits. The length of the current CPU visit is computed every time the process obtains the CPU with:

$$t_{cpu} = \frac{p_{remaining\ cpu\ time}}{p_{remaining\ io} + p_{remaining\ pages} + p_{remaining\ user} + 1}$$

6.2.5 File IO Access Patterns

The only file IO data available from the accounting log is the total number of bytes transferred. There is no data as to when the file IO accesses were performed, how many bytes were transferred during each access, or even if the access was to a local or remote fileserver. Modelling these is difficult as they all depend on either the user, the process or the compute server.

Block Transfers

It is reasonable to assume that there is a minimum physical block that is the actual unit of transfer from the physical device to a memory buffer. Each compute server specifies a standard transfer size, currently an arbitrary two kilobytes. Therefore the total number of physical file IO transfers by a process p is:

$$p_{total\ io\ blocks} = \left\lceil \frac{p_{io\ bytes}}{Transfer\ Size} \right\rceil$$

6.2.6 The Paging Model

In Unix, the paging system is used to load the code required for the execution of a program, and is also used on occasion, for mapping files into the address space of a process. This is termed *background paging* as it is required for the execution of the process, rather than for the management of the compute server's memory, termed *forced paging*.

In extreme cases *forced paging* will degrade the performance of the system as a whole, but in practice, these situations are rare. However, forced paging still needs to be modelled in the simulator, as it represents memory resource contention, a potentially important factor in the evaluation of the *process mix* hypothesis.

Thus the paging activity in which a process is involved consists of two distinct components:

- **Background paging:** For bringing in code, files, etc. in the normal execution of the process.
- **Forced paging:** The memory system forces pages held by the process to be reclaimed for use by other processes.

The pages requested by the process are logical pages, while the pages transferred from a device to memory by the system are physical pages. The information on the number of logical or indeed, physical pages transferred is not recorded by the accounting logs. The physical paging performed by a single process can be measured with the Unix command *time*. However, time only reports the physical pages transferred. The difference between the number of logical and physical pages transferred is due to:

- **Code sharing:** Code or read only pages may be shared when processes using the same binary are coresident. In this case, logical requests do not result in a physical page transfer.
- **Page Reclamation:** Pages that have been added to the free list are able to be reclaimed, and therefore do not constitute a true physical paging event.

These features of a real system obscure the logical paging that was carried out by individual processes, and modelling these features is difficult. To simplify matters, the simulation models logical paging, where all logical requests result in a physical page transfer.

Assumption 1 (paging) *There is a one to one correspondence between logical (requested) and physical (actual) paging.*

Once a paging event has been triggered, it is serviced by the local file IO system as a normal file operation, see section 6.3.4.

Estimating Background Paging

To model background paging, it needs to be related to some known quantity. The most obvious choice is as a proportion of the program's binary file size, as the background paging reflects the loading of the program. However, as the binary file also contains infrequently executed code such as, exception handlers, only a proportion of the binary file should be paged in.

To determine a reasonable value for this proportion, the paging from a small set of diverse test programs was measured on a single SUN IPX machine. All users were excluded from the machine for the duration of the measurements, to avoid the prospect of code sharing and page reclamations. Page reclamations between presentations of the test set were avoided by rebooting the server between the test runs. Due to the rather long cycle between experiments, only a small number of measurements were taken and are summarised in table 6.1. Each program ran to completion except emacs, which was run until the user could begin typing. The particular programs used to measure background paging were selected arbitrarily to represent a cross section of process types.

program	samples	page faults	bytes paged	filesize (bytes)	average ratio	std deviation
sim	4	10.50	43008	122880	0.35	0.03
pasC	4	8.00	32768	106496	0.31	0.00
xteddy	4	48.75	199680	245760	0.81	0.04
emacs	4	280	1146880	2334720	0.49	0.02
overall					**0.49**	**0.20**

Table 6.1: *Paging as a proportion of file size.*

There is only a small variance between the runs of each of the programs, and the final average is larger than two standard deviations. These results indicate that a background paging rate of *half* the binary file size is a reasonable estimate, and this will be used in the simulation.

The *page size* on the SUN IPX is 4 kilobytes, and is the same as that used in the simulator. Thus the number of pages brought in during background paging for a process

p is:

$$p_{total\ pages} = \left\lceil \frac{p_{file\ size}}{2 * page\ size} \right\rceil$$

The stochastic distribution of page references detailed in section 6.2.3 is somewhat contrary to the traditional single process paging model presented in Deitel [Dei84] which is shown in figure 6.2.

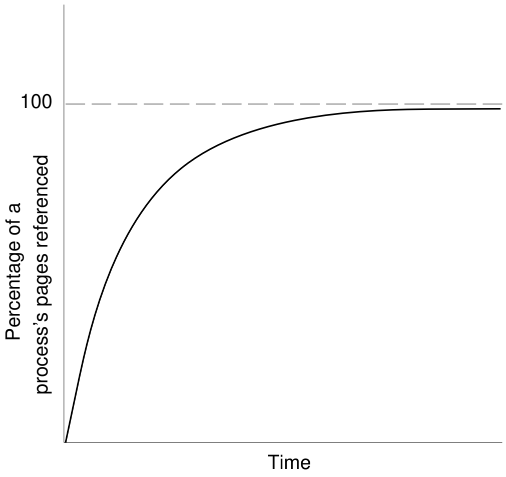


Figure 6.2: *The percentage of a typical process's pages referenced from the time the process begins execution.*

However, the stochastic model is useful as it eases the modelling of forced paging activity, and although it is inconsistent for a single process, on average the paging across all processes on a compute server will average to produce a similar degree of paging activity to the stochastic model.

6.2.7 Estimating Forced Paging

A process may be required to do more than just background paging when there is a shortfall in available physical memory. Modern memory management systems are good, and there is usually little forced paging, however there are extreme situations where forced paging does occur. To determine the magnitude of this degradation, the response of the virtual memory system on a standard Sun IPX running SunOS 4.1.3 was subjected to intense overload, and the results were monitored with the Unix command *vmstat*[6].

[6] Vmstat reports amongst other things, the activity of the virtual memory system.

The Environment

The experiments were again run on a Sun IPX with no other users. Code segment sharing was prevented by giving each process its own copy of the binary.

The Experiment

The program selected to exercise the paging system was a modified version of *lastcomm*[7], extended to gather statistics. The program featured a high degree of file access from reading the logs, a large hash table used to store the statistics - giving the program a very large working set size, and a small binary file size of 35K

The performance of the paging system was monitored with vmstat set to the fastest update rate of 5 seconds. Typical output of vmstat is shown in figure 6.3.

procs			memory				page					disk				faults			cpu		
r	b	w	avm	fre	si	so	pi	po	fr	de	sr	s0	s1	d2	d3	in	sy	cs	us	sy	id
10	0	0	0	364	0	0	8	44	48	0	6	3	1	0	0	44	2441	39	22	76	2
10	0	0	0	344	5	0	8	36	44	0	9	3	0	0	0	45	2424	37	18	79	3
10	0	0	0	292	0	0	0	36	48	0	8	1	0	0	0	32	2476	31	20	77	2
10	0	0	0	300	0	0	0	36	44	0	13	2	0	0	0	34	2450	31	22	74	4
10	0	0	0	284	5	0	0	16	24	0	17	2	1	0	0	34	2448	34	20	77	2
10	0	0	0	296	5	0	0	12	24	0	17	0	1	0	0	26	2489	32	18	78	3
10	0	0	0	268	5	0	0	0	20	0	28	0	0	0	0	20	2514	33	20	77	4

Figure 6.3: *Sample output from* **vmstat**, *the interesting fields are fre (free list in kilobytes), pi and po (kilobytes paged in and out) and fr (kilobytes freed by the paging daemon).*

As the amount of free memory is not under direct experimental control, the best that can be achieved is a relationship between the free memory and the amount of memory paged. In order to sample a reasonable cross section of paging system performance, experimental runs were taken with five, ten, fifteen, twenty and twenty five concurrent copies of the test program.

The paging in and out totals from each run were summed, normalised and averaged against the other runs. These measurements were repeated over three days and the results combined to give figure 6.4. The runs with five and ten concurrent test processes ran to completion, while the larger degrees of concurrency involving fifteen, twenty and twenty five processes were terminated by the system before finishing. The terminations

[7] A UNIX utility for displaying the previously executed commands from the accounting logs

probably resulted from the inability of the compute server to hold all of the working sets in memory and a general shortage of virtual memory, and thus only the runs for 5 and 10 concurrent processes are used to determine the effect of a physical memory shortage.

The paging shown in figure 6.4 is only the low memory paging, and should consist of mostly forced page transfers. Several factors contribute to this statement. Firstly each test run was started on a freshly rebooted machine, so the paging in of the code binaries is at a much higher free memory value. Secondly, the test program grows very slowly, meaning that all of the test processes started well before memory was in crisis. These lead to the conclusion, that most of the paging activity pictured is forced.

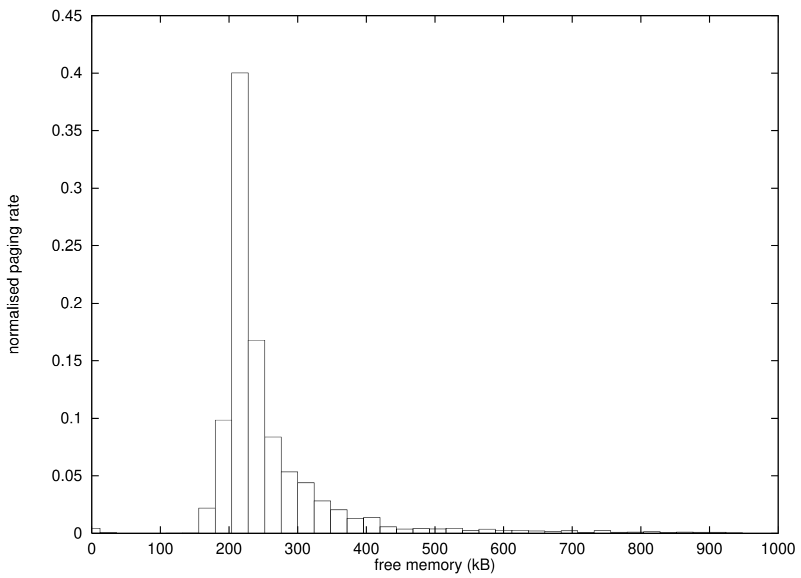


Figure 6.4: *The paging rate as measured from an overloaded system with 5 and 10 concurrent test processes. The area under the bars sums to 1, giving around 40% of low memory paging between 200 to 225 kB. The source of interest in this graph is the increase over the background rate as the amount of available memory decreases.*

The Sun 4 Paging system

The SunOS 4.X paging system described by Cockcroft [Coc93] uses three values to control the paging rate in the system. Firstly; *lotsfree* is the threshold (256 kilobytes) below which the paging daemon is invoked; *desfree*, which is the desperation level, a point above which the paging daemon tries to maintain free memory (100 kilobytes); and lastly *minfree*, which is the minimum free memory (32 kilobytes) considered tolerable before involuntary swapping begins. The scanning rate of the paging daemon increases linearly below *lotsfree* as shown in figure 6.5. A consequence of these thresholds is that all forced paging in a system will occur below *lotsfree*.

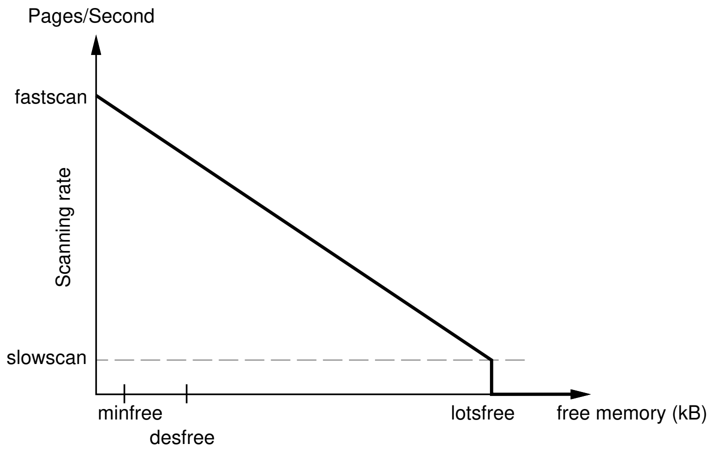


Figure 6.5: *The parameters that control the page scanning rate in a SunOS 4.X system.*

Vmstat

There are considerable limitations to the data provided by vmstat, firstly *fre* is an average over five seconds, and secondly the paging figures are of the kilobytes transferred in only the last second - thus losing all but the most recent value. The averaging of the free list prevents direct demonstration of the SunOS paging mechanism as any memory shortage will usually be resolved before it can have any noticeable effect on the average. The results obtained via vmstat are still useful in building a model however, as the proportion of paging is still much the same even with missing paging activity and as the simulator uses average memory figures, averaged free memory is sufficiently accurate.

Analysis

The frequencies of forced paging in figure 6.4, agree with the action of the paging daemon described in the previous subsection. Essentially, even though the page scanning rate increases linearly freeing more pages, the pages freed are more likely to be needed by the process from which they were reclaimed. This would indeed result in the non linear growth observed.

Simulation Values for Forced Paging

The forced paging in a system can be modelled on a per process basis, namely the effect of a physical memory shortage is to increase the rate at which processes must page by altering the background rate. Figure 6.4 indicates the rate of paging increase as the amount of physical memory decreases. This rate can be approximated by a multiple of the background rate bg, over certain ranges of free memory, as in table 6.2.

free memory (kB)	rate
> 450	1 * bg
400-449	2 * bg
350-399	4 * bg
300-349	8 * bg
250-299	16 * bg
200-249	32 * bg

Table 6.2: *Paging penalties as memory approaches the desperation threshold.*

Free memory values below 200 are not permitted in the simulator to represent the minfree threshold where swapping occurs.

Increasing the rate of paging will simply cause a process to do all of its paging at the start, as pages brought in are subtracted from the total number of pages required by the process. In this case, a notion of *paging success* is required, where a successful page transfer is one that does not result in the paging process losing another page before it next claims the CPU. The arbitrary values in table 6.3 are intended to represent the rate of success that a process meets after paging at a low memory availability.

free memory (kB)	rate	Pr(suc)
> 450	1 * bg	100
400-449	2 * bg	85
350-399	4 * bg	65
300-349	8 * bg	45
250-299	16 * bg	28
200-249	32 * bg	10

Table 6.3: *The probability of a successful page transfer decreases with memory availability.*

Memory Allocation

When a process starts on a compute server, it consumes the average memory value recorded by the accounting log. This amount of memory in use does not change dynamically in relation to the the paging activities of the process as there is no direct link between the memory and the paging models. It is unclear that integrating both models would be beneficial, especially in light of the increase in complexity.

6.2.8 The User Model

There is a class of processes delayed by factors other than the speed with which they can be provided system resources. This class typically includes *interactive* processes, that must wait on user responses, and *periodic* processes, which are activated by timed events. I will call these processes *externally delayed*, as the source of the delay is independent of the state or power of the machine.

Externally delayed processes have been ignored in previous simulation studies, but greatly effect the speed with which processes progress through the system. If the external delay is omitted from the simulation, the time spent by processes within the system is too short, decreasing memory residency and therefore contention for other resources as well. Counterintuitively, it does not produce the same effect as the infinite memory assumption, as longer stay, externally delayed processes have a greater chance of causing contention with more processes than if they had completed their execution and departed.

Estimating User Time

Given that external delays are significant and require modelling, a model is required that copes with both periodic and user based delays. However, there is no record in the system log on how much time was spent waiting on user responses or timers. From this point, for brevity I will refer to the time spent waiting on all external delays as *user time*.

The obvious approach of subtracting an estimated time for performing the processes CPU, IO and paging, from the elapsed time is unworkable due to extremely variable elapsed times. Code sharing, page reclamation, caching, and the system load are all factors contributing to this variability, of which the net result is, that many known interactive processes appear to have negative user times, a quite unrealistic situation.

A Better User Model

Broadly speaking, processes can be classified into one of two categories, there are the externally delayed processes identified earlier, and then there are *batch* processes which will proceed through the system as quickly as resources allow. Thus if batch processes are party to the same degree of caching, queueing time, code sharing and page reclamation as externally delayed processes, then the only difference between the two types is the user time.

Assumption 2 (process behaviour) *Batch and externally delayed processes exhibit similar behaviour except that batch processes have no user time.*

Thus, if a model can be constructed for batch processes relating known parameters from the trace such as CPU and IO to the elapsed time, then this model can be used to compute the estimated *batch time*[8] for an externally delayed process. The difference between this estimated time and the actual elapsed time is the user time.

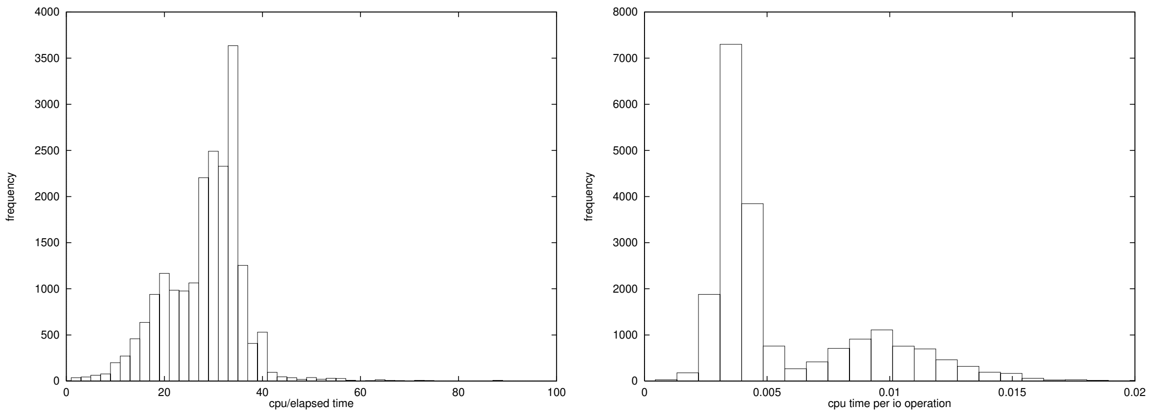


Figure 6.6: *The distributions of df, the leftmost is a measure of how CPU bound the process is, the right, how IO bound. Note the bimodalily, reflecting two different behaviours depending on the command line option.*

Figure 6.6 shows the distributions of these times for the Unix command *df* [9]. The batch job processes chosen to construct the model must have enough samples (between ten

[8]That is, the time the process would take if it were a batch process.
[9]df displays the free disk space statistics

thousand and twenty thousand), and produce consistent runtime behaviour that does not rely on an external component such as the file size, as does *cp*. These criteria result in only five useful candidates giving a total of six points (bimodal = 2 points). Plotting distribution medians produces the relationship f^{10} shown in figure 6.7.

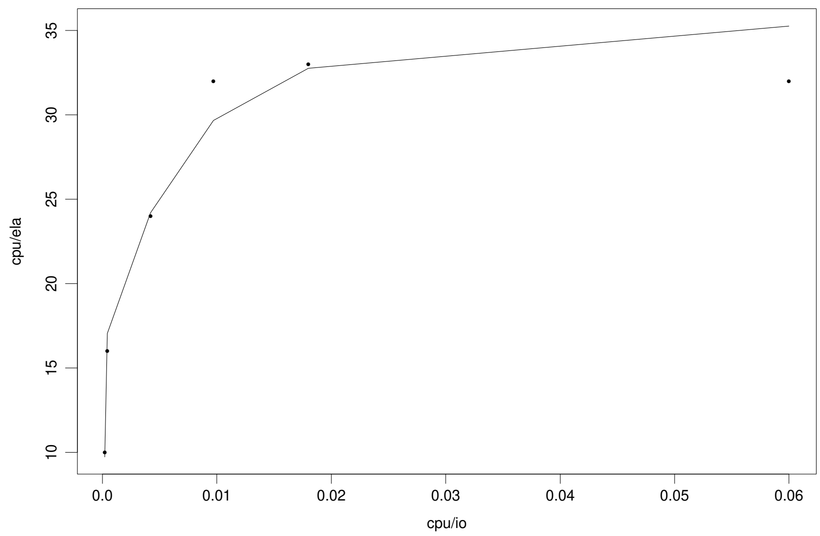


Figure 6.7: *Relationship f between IO-boundness and CPU-boundness.*

Thus, the estimated batch time of a process is computed by the ratio of the CPU and IO times:

$$estimated\ batch\ time = f(\frac{cpu}{io})$$

giving the mean of the CPU-boundness. To preserve some variation, the final value is picked randomly from a normal distribution with the mean set to the value and the standard deviation set empirically to one third of the mean.

User Delay Lengths

Once the user time has been estimated, another model determines how the time is used over the lifetime of the job. The basic assumption is:

Assumption 3 (User Delay Length) *A greater the proportion of user time to total time, gives a longer delay.*

[10] Approximated with $log(x \times A)B$, where $A = 12000$ and $B = 6$

The simulator recognises two types of user delays, keystroke delays, and those due to think times. The keystroke delays where measured from two students typing quickly giving a distribution with a mean μ. To model the longer think times and coffee breaks, the number of user events before the next event of any other type are computed from the probability, see section 6.2.3, generating a new mean value μ':

$$\mu' = \mu \times \frac{Pr(user)}{1 - Pr(user)}$$

If μ' is the same or less than μ, a keystroke delay is randomly selected from the measured distribution, otherwise the μ' forms the median of a new distribution from which the longer user delay is taken. The user delays in figure 6.8 are produced by the simulator with this model.

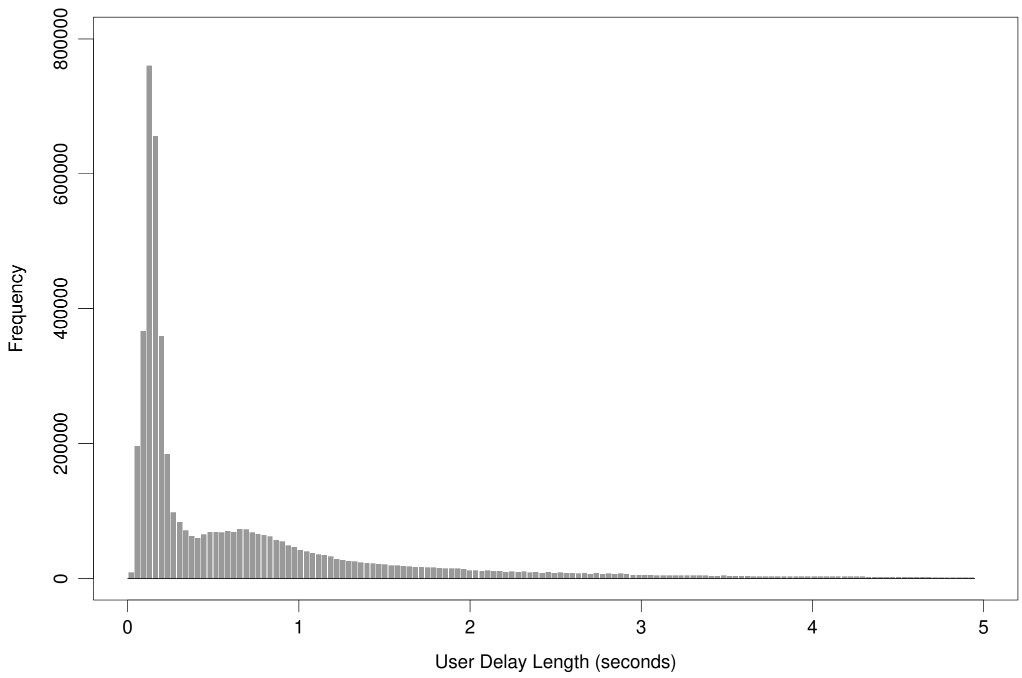


Figure 6.8: *The distribution of user delays generated by the user model. It is bimodal, the first mode at ≈ 110 milliseconds represents the pause between keystrokes in normal typing and the second mode at ≈ 650 milliseconds represents the longer thought pauses. The tail on this distribution is very long to include the possibility of coffee breaks, conversations etc.*

6.3 Compute Server Modelling

The three fundamental resources modelled in the compute server are CPU, memory and file IO. In addition to the provision of these resources, each compute server is also responsible for the sharing of them between competing processes.

The compute server model is designed to allow a considerable degree of resource heterogeneity in both availability and rate of service. A configuration file is used to specify the service rate of the CPU, the length of the quantum, the local and remote file IO service times, and the amount of physical memory. The cost of state collection can also be adjusted.

Processes flow individually from the ready queue into the CPU, which then provides CPU time until the process requires some other system resource, or has its quantum expire. Processes moving to other system resources are returned to the end of the ready queue after servicing, as seen in figure 6.9. The actual branches are followed probabilistically, a point detailed in section 6.2.3.

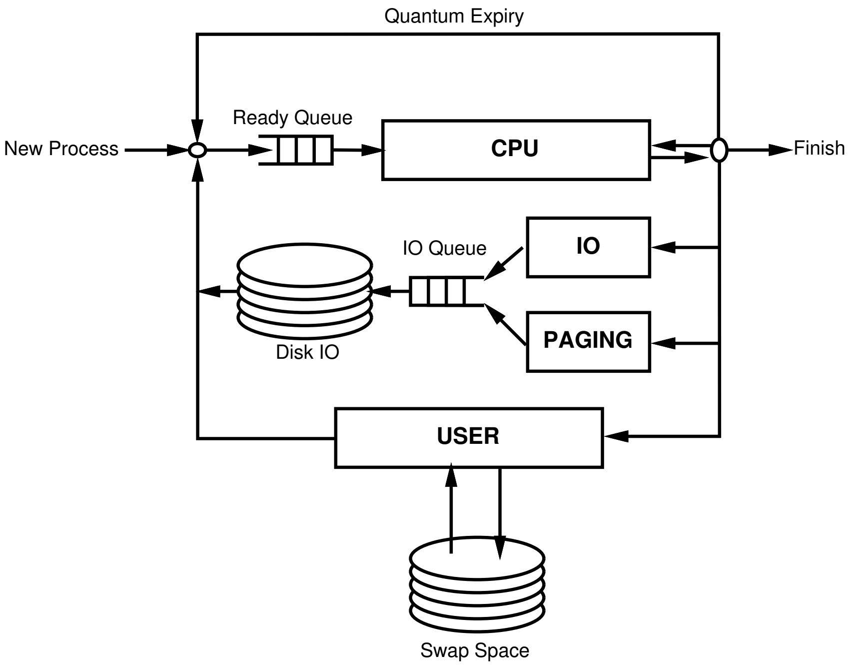


Figure 6.9: *The compute server model. Each process decides which resource to visit, and the compute server shares the resources between competing processes. The IO subsystem illustrated is simplified, showing only the local system.*

Each compute server maintains an independent microsecond clock, used to sequence events within each compute server and over the distributed system. The server clocks are synchronised by the arrival of new processes and the migration of processes.

6.3.1 The CPU

The CPU is the core of the simulator, and all processes must pass though it before entering any other service. Thus the CPU controls the advancement of time within each compute server, and is in one of two states, idle or running. In the running state the CPU advances time to the next event, be it a service request, page fault or quantum expiry, and allocates the elapsed time towards the current process's CPU requirement. In the idle state there are no processes available to run, and in this situation the clock is advanced immediately to the next event, when either a new process enters the compute server, or a process returns to the ready queue from a resource.

Adjustable CPU Performance

The rate at which the CPU can service processes is specified relative to the service rate of the CPU on the machine from which the trace was recorded. Thus if a CPU service rate is specified as 2.0, the CPU will service processes twice as fast as the original machine. All experiments in this thesis were conducted with a CPU service rate of 1.0.

CPU Time

The CPU time of a process is divided into $n + 1$ *chunks*, where n is the total number of visits that a process needs to make to the other resources (IO, paging and user). Each visit to the CPU consumes one CPU chunk, after which the process decides if it wishes to access another system resource or continue with the CPU. If the process reenters the CPU, the chunk size is recomputed to ensure that there is always a CPU visit before any other service. If the quantum expires, the process returns directly to the ready queue. For more detail, see section 6.2.4.

Scheduling CPU Processes

To provide a realistic allocation of processes to the CPU, a scheduling system is needed. There are a number of simple methods of single CPU scheduling that can be considered as candidates for the scheduler, including first come first served (FCFS), shortest job first (SJF), round robin (RR), and multi-level feedback queue scheduling. Round robin and the multi-level feedback queue are both preemptive systems and therefore preferable candidates for simulating modern interactive computer systems. The multi-level feedback queue is likely to add unnecessary complexity to the simulation, as the provision of fast response times for interactive users is not a priority. Thus processes are scheduled in a RR fashion for simplicity, and preempted on the expiry of an adjustable quantum (arbitrarily chosen as 50 milliseconds for all experimental work).

6.3.2 The Memory System

The memory system is often neglected in load distribution simulations with the assumption of infinite memory being common, as in Eager et al. [ELZ86b] and Harchol-Balter and Downey [HBD95]. This is acceptable, if the load distribution algorithms concentrate on the CPU as the basis for load distribution. However, as the resources of the system are vital to full test the *process mix* hypothesis, careful modelling of this primary system resource is required.

Memory Data

The memory information available from the trace is an estimate of the average virtual memory used by a process over its lifetime. This is rather limited as it gives no insight into the rate at which memory was acquired by the process, or the extent to which it could be released in times of memory drought. Figure 6.10 shows the memory that is recorded as allocated in the accounting log on a typical day on a machine which is *not* overloaded. This graph shows that there are periods during the day, when the amount of allocated memory reported exceeds the combined capacity of the swap space and the physical memory. Two conclusions can be drawn from this:

- Code sharing is not accounted for in the log, and is a significant factor.

- The memory allocation record is unreliable, and should be treated with caution.

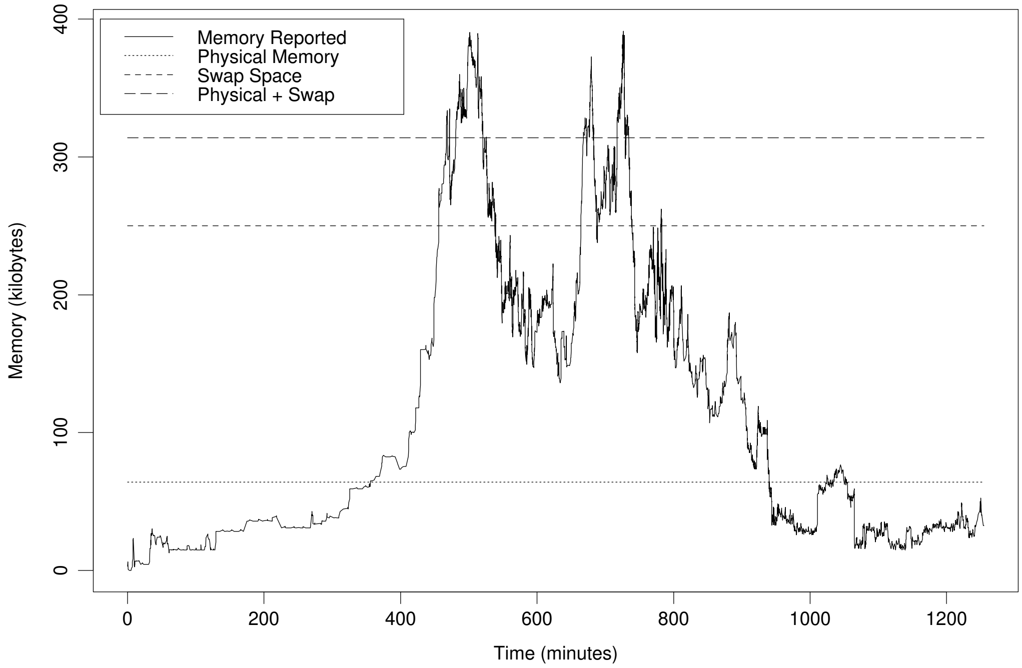


Figure 6.10: *The amount of allocated memory claimed by the accounting log for Cantina on the first of May 1995. The actual period is from 2 am to 10 pm, with 600 minutes corresponding to 12 noon.*

Memory Model

The system memory model must interact with the processes in an intelligent way, yet remain simple. As there is no dynamic memory data available, I will adopt a static memory allocation model, where the amount of memory recorded for a process is allocated to that process for its entire lifetime. This immediately leads to the observation from figure 6.10 that physical memory will be completely allocated from eight fifty in the morning until eight in the evening. It is obvious that some means is required for extending the availability of physical memory in the system, and the associated management of competing processes. I will neglect the significance of code sharing, and allow an infinite swap space:

Assumption 4 (code sharing) *The absence of code sharing will affect all load distribution algorithms in the same way, and therefore is not essential to the model.*

Assumption 5 (swap space) *A Swap space is infinite.*

An infinite swap space is a reasonable assumption, as none of the traces recorded a compute server being so overloaded that it could not start a process.

6.3.3 The Memory Swapper

Physical memory is a finite resource just like CPU time, and therefore must be shared between competing processes. Ready processes and those waiting on the CPU or IO queues must be *active*, while processes waiting on user or external events, as described in section 6.2.8, may be *inactive*. Active processes require a presence in physical memory, while inactive processes may be swapped out.

When there is a physical memory shortfall, processes that are waiting on user or external events may be removed from memory and placed in the swap space. Also, if there is no memory available to start a new process then it must still be initiated, but placed in the swap space until there is room to begin execution.

Swapping Out

The decision of when demand is sufficient to result in exile to the swap space is an arbitrary two tiered scheme and swapping is initiated when either:

- Available physical memory falls below 10%, at which point any process beginning a user delay of greater than one second is swapped out.
- The process is scheduled for a user delay longer than five seconds, in which case it is always swapped out.

The process will stay in the swap space until it has completed its delay period, and it then becomes eligible for returning to physical memory.

Swapping In

Swapping in is accomplished by placing eligible jobs onto a priority queue, ordered by the time of eligibility. When memory becomes available (another process finishes, or is swapped out), then processes that are small enough to fit are removed from the front of the queue. With this heuristic it is possible that large processes will accumulate at the front of the queue, and will suffer starvation in times of low memory availability. In practice this does not appear to be the case, and the response times of even very large processes are close to those experienced on the original system.

The Performance of the Memory System

The memory swapper produces very realistic physical memory results, as shown in figure 6.11. The memory utilisation in the simulated distributed system remains high for long periods of the day, with memory being freed only when there is a peak demand, or a process terminates. This produces results similar in effect to the SunOS virtual memory management discussed in section 6.2.7.

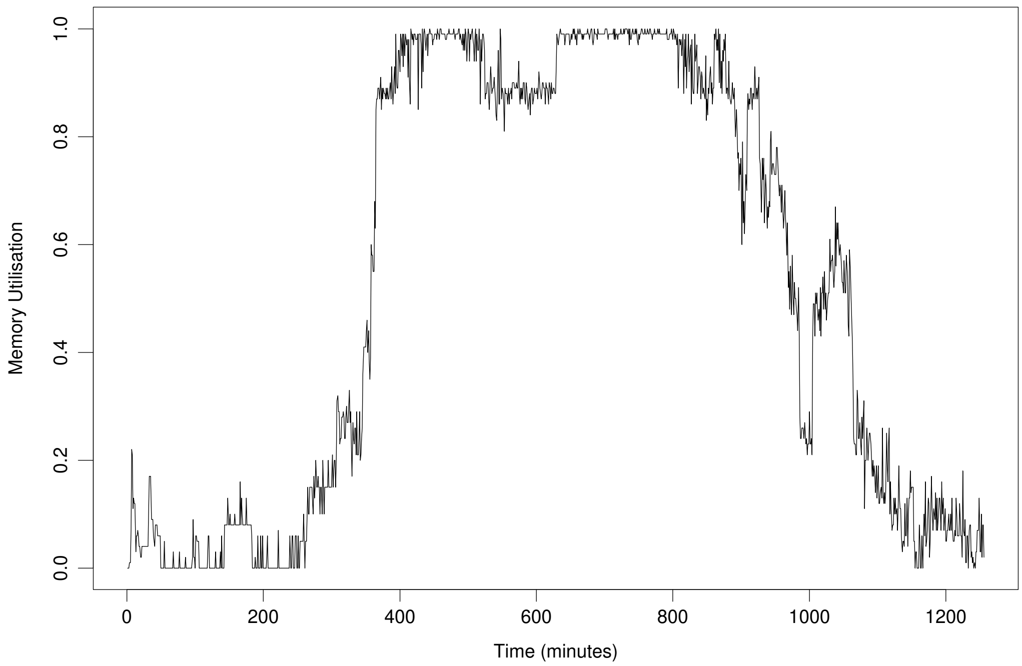


Figure 6.11: *The physical memory allocated in the simulator over the same period as figure 6.11.*

The memory swapper is critical to the simulator if the assumption of infinite memory is to be avoided, and a realistic memory statistic is to be reported for the load distribution algorithms.

6.3.4 The Disk IO Subsystem

The Disk IO subsystem services both the logical file and paging IO requests from processes. Two different sets of assumptions are required for file and paging IO:

File IO Assumptions

There are two types of file IO present in processes running on in a distributed system: the accesses to the compute servers local disk *(local file IO)*, and those that access disks on remote compute servers *(remote file IO)*.

There is no distinction between local and remote file IO in the accounting logs, and therefore some assumptions are required if there are to be realistic costs for local and remote execution:

Assumption 6 (file visibility) *All files are visible to all compute servers, that is, the distributed system is equipped with a transparent network file system.*

This assumption mirrors any real system which has a network file system.

Assumption 7 (file locality) *Files are local to the compute server on which a process is originally initiated.*

As there is no distinction made in the accounting logs between local and remote file IO, any decision must be arbitrary. The assumption that the host on which the job was intended to execute (prior to load distribution) holds all of the required files locally is both simple and reasonable. There are two main implications resulting from this assumption. Firstly, file access will always be less expensive when a process is run locally, and secondly, all accesses made by a single process are of the same type (i.e., all local or all remote).

Assumption 8 (file caching) *There is no file IO caching in the system. Thus every file IO access in the accounting log results in an access to the disk.*

Caching is a thorny problem when processes are redistributed across the system. For example, a migrated process will have to populate the cache on the new compute server, and therefore experiences a low hit rate. A fixed hit rate is misleading, and modelling anything else would add additional complication to the model with little expected benefit.

Assumption 9 (file access cost) *Local read and write operations incur the same service times.*

This last assumption is supported by the measurements made in section 6.3.4.

Paging IO Assumptions

All paging access are to the local disk for processes executing locally and remotely, under the following assumption:

Assumption 10 (local binaries) *Code binaries are mirrored on the local disk.*

The only exception to this ,is when a process has been migrated, in which case all pages are fetched from the remote disk, see section 7 for more detail.

The Design

Local and remote disk IO are fundamentally independent when considering disk IO on a single machine, one is serviced by the local disk, and the other by the network. The immediate observation is that these are parallel events, with no interference between them. The situation is more complicated, when the distributed system as a whole is considered, as every remote disk IO access will result in a local disk IO operation on the compute server that physically holds the file or page.

The simulator models disk IO on each machine with two queues as shown in figure 6.12, and each has a different service rate (see section 6.3.5). The local queue services all local disk IO accesses, and in effect, equips each compute server with one device for local disk IO.

The single remote queue services all remote disk IO accesses made from foreign processes executing on the machine. The distributed system being modelled is based on a set of peer servers, where users local file systems are distributed evenly between the servers, and there are no specialised file or compute servers. In this situation, I suggest that remote disk accesses will be evenly distributed across the network, and as a consequence, it is sufficient to model the contention for remote disk IO devices on a per compute server basis (i.e., the contention for remote disk IO, averages out).

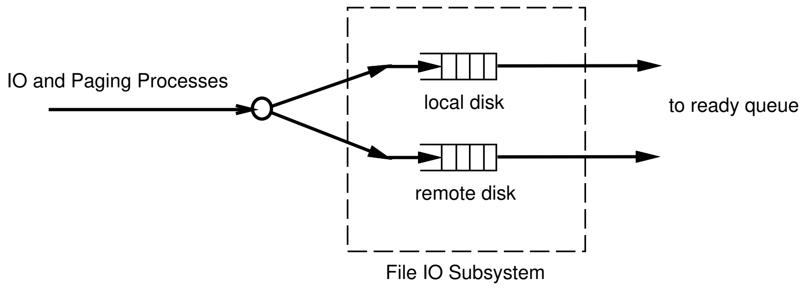


Figure 6.12: *The Disk IO subsystem.*

Both of the disk IO queues operate on a simple first in first out (FIFO) basis, with processes leaving the queue at a maximum rate determined by the service time of the device.

Table 6.4 indicates that for similar disk IO utilisation, the higher the number of seeks, the lower the amount of data transferred. This implies the assumption that:

Assumption 11 (Seeking) *Seeking dominates the time to access and transfer a block from a physical disk.*

Processes performing file and paging IO are treated in exactly the same way by the disk IO subsystem, except that, a file IO transfer is a 2 kilobyte byte block and a page is an 4 kilobyte block. The two different block sizes both have the same disk service time due to the seeking assumption.

6.3.5 Determining Disk IO Performance

To ensure that the disk IO service times are realistic and more importantly that local and remote access times are in proportion, the disk IO performance of a Sun IPX workstation was measured with the Unix command *iostat*[11], and the results are summarised in table 6.4. The disk IO service times for the simulator are chosen as the minimum time measured on the real system. The reason for choosing the minimum

[11] Iostat is a UNIX command that reports the disk activity.

rather than the average service time, is that the model of the disk IO system includes contention for the resource, and therefore the maximum rate at which the device can service requests is the significant factor.

The disk IO performance was measured using the following procedure:

- **Local** disk IO performance was measured using a program that wrote a very large file of random information to a local disk on the test SUN IPX. The read performance was then tested using this file, taking around five minutes to read the file, and producing around 300 samples per run. The cache was flushed between the writing and reading phases of the experiment to ensure that the actual physical transfer time was measured, (see section 6.3.4 file caching assumption).
- **Remote** performance was measured using the same principle, except the test file was written to a remote disk. The remote machine was a DEC Alpha workstation situated on the same subnet as the test Sun IPX and to maintain consistency all results were measured on the Sun IPX.

Local Disk IO							
	maximum access frequency/second			maximum transfer rate/second			
run	accesses	transfer (kb)	disk (%)	access	transfer (kb)	disk (%)	action
1	**82.0**	904	96	62.0	**1425**	94	read
2	77.1	872	94	64.9	1237	95	read
3	80.0	863	96	62.9	1182	95	read
4	81.0	976	99	69.0	1423	97	read
5	**82.0**	984	100	42.0	1376	100	write

Remote Disk IO							
	maximum access frequency/second			maximum transfer rate/second			
run	access	transfer (kb)	disk (%)	access	transfer (kb)	disk (%)	action
1	**63.0**	688	84	51.0	**909**	74	read
2	52.0	487	85	45.0	560	84	write

Table 6.4: *Maximum disk IO performance on a SUN IPX.*

Table 6.4 lists two different sets of results for each run. The first set of three columns refer to the number of accesses made to the disk, the number of bytes transferred and the utilisation of the disk for the one second period that had the highest access rate for that run. The second set of three columns represent the same quantities, except the values correspond to the one second period that had the highest number of bytes transferred. These two results give some support to the seeking assumption from section 6.3.4, as they show a distinct tradeoff of seeking and transfer.

Selection of disk IO Service Times

The maximum of 82 accesses per second on local reads and writes from table 6.4 corresponds to a local service time of approximately 12 millisecond per access. The 63 accesses per second for remote accesses gives a remote service time of approximately 16 milliseconds per access. These are the service times that will be used in the testbed simulator for all experiments.

The Test Environment

The test runs were conducted on a freshly booted system with no other users present. The measurements for writing are included only for completeness, as reads tend to dominate disk IO and hence it is the value for the read operation that is used a basis for the simulator service times.

Observations

There is little difference in the performance of the local read and write operations and these results support the file access assumption, made in section 6.3.4 for local accesses.

The results indicate that remote writing is more costly than remote reading, but there is the possibility of outside interference, as it was not possible to isolate the remote compute server from other users.

The disk IO test program exhibits high access locality, as the single test file is read sequentially. The SunOS 4.X file system attempts to arrange files in rotationally contiguous strips, thus minimising head latency and seeking. However, modern disks perform remapping of logical tracks and sectors to physical tracks and sectors, and such a scheme may no longer perform its desired function. Thus even though the file is logically contiguous it may not be physically contiguous, and a sequential file access may involve seeking and head latency. Thus this is a reasonable approximation to the situation where real processes have more than one file open, and are quite likely to make use of seeking inside them. Even if this were not the case, the important feature is the proportional cost of local and remote disk IO devices, since both local and remote systems experienced the same locality, the proportions are preserved.

6.3.6 Compute Server Instrumentation

Each compute server also needs to collect load information for both load distribution activities and for measuring the performance of the system.

The system load is collected on each compute server by periodically running a *system load process* that also updates the globally accessible state. The information collected by the system load process is:

- CPU utilisation
- Local disk IO utilisation
- Free memory
- Instantaneous length of the ready queue
- Instantaneous length of the local disk IO queue
- Instantaneous length of the remote disk IO queue (awaiting NFS IO)
- The number of processes currently in the system
- A timestamp

The utilisations are taken over a period equal to half the update period τ. This ensures that the utilisation statistics are at most, $\frac{\tau}{2}$ old. All other values are purely instantaneous.

The system load process is treated in exactly the same way as normal processes, and incurs all normal queueing delays. The resource requirements of the system load process are specified in a compute server configuration file, but for these experiments no cost was attributed to either the collection or dissemination of the load information. This lack of cost does not invalidate any comparisons made between different load distribution techniques, as all of the algorithms share the same load metric and communication architecture.

One major difference between the system load process, and all other processes is that it may not be migrated. If it were, the source host would no longer make state updates, and the destination host would make multiple updates.

6.4 Summary

Models of each of the primary resources (or sources of delay) in a compute server have been constructed with the intention of providing a sufficiently realistic distributed system simulation for the evaluation of resource balancing.

The simulation is driven by the processes, which move between compute server resources based on each process's individual requirements, thus creating realistic contention for popular resources.

Remotely executing processes differ from local processes by the service time of file IO operations. Other costs that a remotely executing process may involve are remote displays, message forwarding and in some implementations, all kernel calls must be serviced on the original compute server. In order to preserve simplicity in the simulator, these costs are neglected.

Chapter 7

Load Distribution Mechanism

The taxonomy from section 2.2.3 identified three major components of the mechanism required for load distribution: the load metric, load communication, and the transfer mechanism. These three components are discussed in sections 7.1 to 7.3. Section 7.4 details the mechanism to supply job resource predictions.

The mechanisms developed in this chapter are simple, as the objective is not to test or develop new mechanisms, but rather to develop and test the policies which they support. Thus for simplicity, the mechanism is implemented with a global load collection and local policy modules.

7.1 Load Metric

The load metric, provided to all[1] of the load distribution algorithms in chapter 8, is a three dimensional vector of CPU, memory and IO utilisation.

$$\mathbf{load} = < CPU_{util}, memory_{util}, IO_{util} >$$

This load vector is further refined by each load distribution algorithm to provide its desired type of ranking. For example, the *standard* metric used in the simulation is the average of the vector components[2]:

$$\overline{\mathbf{load}} = \frac{CPU_{util} + memory_{util} + IO_{util}}{3}$$

[1]To ensure fairness in the evaluation of the load distribution policies, all must have access to the same information on which to distribute the workload.

[2]This is in effect a linear combination which was found to give good performance by Ferrari and Zhou [FZ87].

The resource based algorithms use the three vector components individually, as explained in section 8.3.1.

7.1.1 Resource Utilisation

Resource utilisation values are used in the load vector, as unlike the queue length, the range over which they occur is well bounded. This behaviour is important for the resource based algorithms, and to ensure comparability and fairness, all algorithms employ utilisation.

7.2 Load Communication

The exchange of load information is essential for dynamic load distribution. Unfortunately collection of load information is often expensive, and too frequent exchange of such information can result in unacceptable communication overheads. A number of methods for the collection and exchange of load information were described in section 2.3.6. Of these, the two most common are *polling* and *periodic updating.* With polling, all potential hosts in the system are contacted before each job is scheduled, and a current load vector assembled in reply. This could result in the job being delayed while waiting on the other hosts to respond[3], and as the number of job arrivals in the system increases, there will be a corresponding rise in the network traffic. Periodic updating is cheaper, and when cached on the local machine, is fast. The major problem is that information can become stale between updates, and result in incorrect job placements[4].

The architecture of the information exchange mechanism involves a local (per host) update process that forwards the load vector to a global load collection agent (GLA) at time T, as shown in figure 7.1. The GLA distributes this information to each host after the last update is received at time $T + \alpha$. The load distribution algorithms then use the local copy to make load distribution decisions.

The only exceptions to this design are the load distribution algorithms that estimate the load between periodic updates. In these cases, the new load estimate must be forwarded

[3] If complete information is required. If only the first one or few replies are needed this is less of a problem.

[4] Process migration does not suffer from this problem as it is initiated immediately after the load update, and therefore the load data is used before it becomes stale.

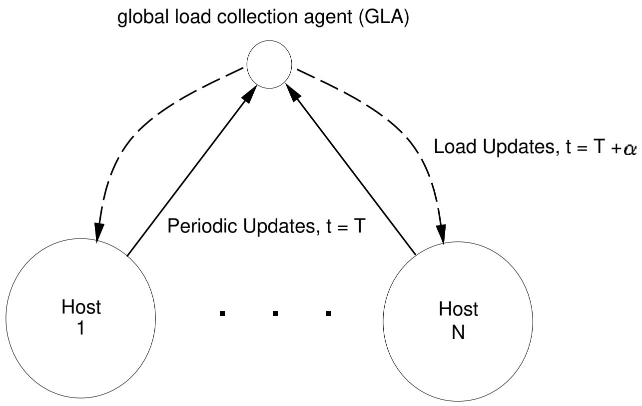


Figure 7.1: *Global state update collection and distribution.*

to the GLA, which in turn must relay this estimate to all hosts.

7.3 Load Transfer Mechanism

As detailed in section 2.3.4, load distribution can occur either before a job is started, or during its execution. The mechanisms to perform these two types of transfer are distinct, and implemented separately on top of the distributed system simulation. The following two subsections, detail each of these transfer mechanisms, and the order in which transfer actions occur.

7.3.1 Initial Placement

Initial placement transfers jobs before execution, consequently requiring little extension of the distributed system simulator. Figure 7.2 shows the sequence of events from **a** to **e** required to initially place job j at time t, for all algorithms except *random*.

Initial placement is triggered by the arrival of a job j, on the source host at time t (**a**). The relevant load distribution policies are used to determine if the job is suitable for remote execution (**b**), and if so, where it should be executed. If the load distribution policy calls for the load to be estimated between periodic updates, the load is updated via the GLA (**c**). The transfer of j to the destination host in step **d** is modelled using a delay queue, in which each job intended for remote execution is suspended. The length of the suspension is fixed at some time T, regardless of the job size and destination host. Job j finally begins execution in step **e** on the destination host at time $t + T$, where T

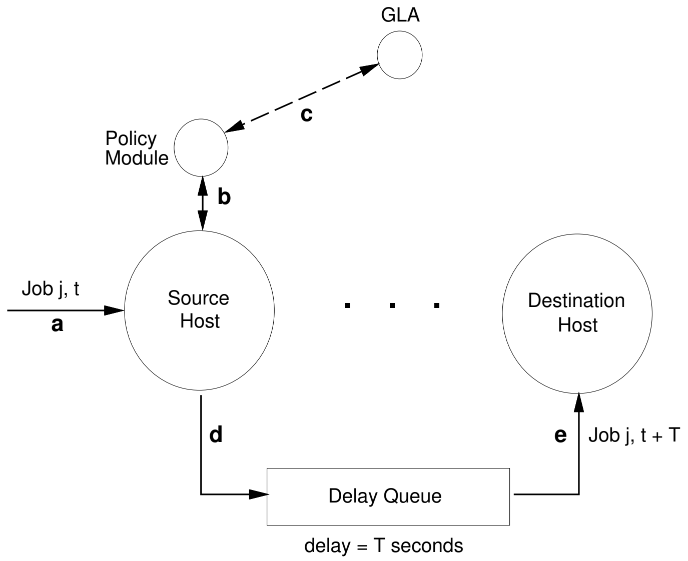


Figure 7.2: *Sequence of events from* **a** *to* **e**, *to initially place job j.*

represents the entire cost to the system, as discussed in section 7.3.3.

7.3.2 Process Migration

A process may be migrated at any stage during its lifetime, therefore it would be possible to continuously consider each process as a potential migration candidate, however, Barak and Shiloh [BS85] identified this continuous consideration as an unacceptable system overhead. The simplest approach is to constrain the system to consider migration periodically, immediately after receiving the global load update from the GLA, in order to take advantage of the freshest information. Figure 7.3 shows the sequence of events for migrating a process p within the testbed simulator.

The global load update at time t from the GLA (section 7.2) triggers each host to consider migration in step **a**. The host with the highest load (see section 7.1) selects a suitable process if there is one, and marks the process as *to migrate* in step **b**. The mechanism can only migrate processes from the ready queue, (described in section 6.3), and therefore if a marked process is in any other state, it will not migrate until it next returns to the ready queue after a delay of α. The mechanism does not place any constraints on which processes may be migrated, it simply determines when migration can physically occur. Avoiding candidates such as those which will not shortly return to the ready queue, is the prerogative of the candidate selection policy, not the mechanism.

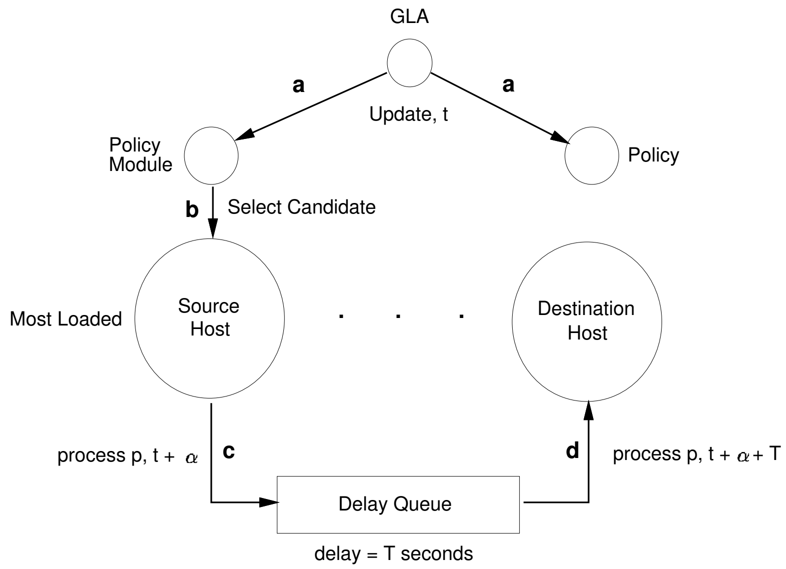


Figure 7.3: *Sequence of events from* **a** *to* **d** *to migrate process p.*

Once migration has begun, the migrating process p enters the delay queue in step **c** for the same period as initially placed processes. During the delay period, p is in the *migrating* control state, where it is still present in the source host, but frozen. After the delay period, p is inserted into the ready queue on the destination host (**d**), it's current runtime state is updated to suit the new location, and p finally continues with it's interrupted execution after a delay of $t + \alpha + T$. The delay period T does not account for the cost of transferring the address space to the new host, instead this is done with the paging mechanism as described in the following subsection.

7.3.3 Cost of Transfer

Previous researchers, Leland and Ott [LO86] and Harchol-Balter and Downey [HBD95] for example, have described the cost of migrating a process p, as the sum of some fixed overhead (T), and a variable cost of moving p's address space. Whilst the variable cost is extended to include terms such as the cost of moving open files in [DO91], the basic model is sufficient for the granularity of this simulation.

$$\textit{migration overhead} = \textit{fixed cost} + \textit{memory transfer cost}$$

The 'fixed' part of the fixed cost comes from the fact that the cost is mostly independent of the job being transferred rather than it being fixed for all system and network loads.

In the testbed simulation, the fixed cost is reflected as a delay in the execution of a process. There is no CPU, memory or IO load imposed on a host for suspending, packaging, transferring or restarting a migrated process. Initial placement does not transfer processes, so there is no corresponding cost for transferring an address space. Consequently the cost of initial placement is just the fixed cost.

- **Fixed Cost (T):**

  The fixed cost is highly variable over a range of real systems and implementations. For example, it takes Sprite [DO91] 330 milliseconds to transfer a null job, while it takes *rsh*[5] [Hol96] about 2.1 seconds. This variation suggests that it would be better to consider a range of fixed values rather than selecting a value arbitrarily.

- **Address Space Cost:**

  In the testbed simulator, the memory transfer cost is not translated into a simple delay, as is the fixed cost, but modelled using the paging system described in chapter 6. Essentially, when a process p is migrated, the amount of paging it needs to perform is reset to:

  $$p_{required\ pages} = \left\lceil \frac{p_{addr\ space\ size}}{page\ size} \right\rceil$$

  This is a worst case scenario, where every page from the process is transferred. The cost of transferring a page is not the normal local cost of 12 milliseconds incurred for all other paging, but the remote IO cost of 16 milliseconds, as obviously the *local binaries* assumption from section 6.3.4 does not hold for address space data.

  This cost model results in a raw memory transfer rate of approximately 650 K bytes/second. Interestingly this is close to the 660 K bytes/s rate required by the Sprite system to flush its dirty pages to the network page server.

7.3.4 The Update Process

It is worth noting that all load distribution policies avoid selecting the update process (see section 6.3.6) as a load distribution candidate. If this were to happen, the source host would no longer send updates to the global state collection agent, and the destination host would send two copies.

[5]Rsh is a Unix command for remote execution.

7.4 Prediction Mechanism

It is clear from chapter 3, that if initial placement is to be used to maintain the *process mix* on a host, then some form of *a priori* knowledge about the jobs being scheduled is required. The most useful way to provide this information is by forming predictions based on historical behaviour, with a probabilistic rule, as in Harchol-Balter and Downey [HBD95], or learnt behaviour as in Koch et al. [KRK94].

7.4.1 Taxonomy Extension

A prediction mechanism is not a traditional part of load distribution, and most previous work has not considered any form of *a priori* knowledge. Essentially, a prediction is simply extra information on which a policy may choose to act, and the insight it adds, is the job's potential contribution towards the load on the system, and as such, can be viewed as an extension of the load metric. Thus the logical extension of the taxonomy from section 2.2.3 is to split the branch containing the load metric into branches for host and job metrics, as shown in figure 7.4.

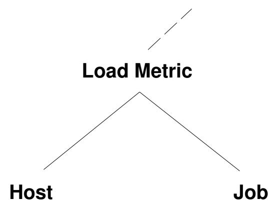


Figure 7.4: *Extension of the taxonomy to include prediction.*

7.4.2 Implementation

Chapter 4 included several techniques used to predict the future activity of jobs. The predictions supplied by the mechanism described in this chapter are based on the running average of resource use from past executions. The four values recorded for each program are:

- Average CPU time
- Average Memory allocation
- Average IO transfers

- Average Duration

The physical implementation uses a global hash table indexed by the program name to locate the past average resource requirements for each program. The interaction of the prediction mechanism and load distribution system is shown in figure 7.5.

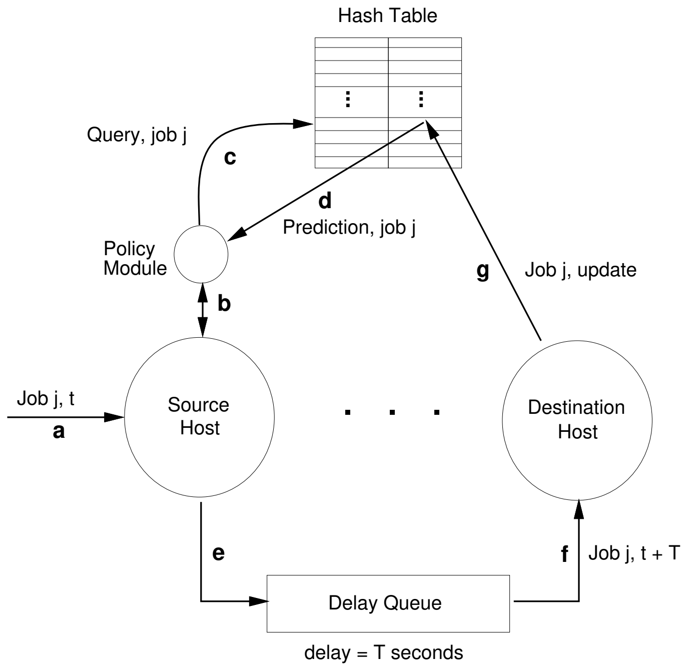


Figure 7.5: *The prediction mechanism and its interaction with the initial placement mechanism.*

Initial placement is triggered by the arrival of a job j, on the source host at time t (**a**). The policy module is contacted (**b**) which performs a lookup on j (**c**) and receives j's past average resource requirements (**d**). Using this information, the job is scheduled to a destination host **e**, where it starts execution at time $t + T$ (**f**). After j has completed (in the finished control state), the resource requirements from this run are forwarded to update the average in step **g**.

The time spent in the delay queue T, is not counted towards the average duration as it is a scheduling artifact rather than behaviour inherent to the job or host.

The duration is dependent on the characteristics of the job as well as the load of the host on which it was executed. Unlike the delay T, this is difficult to compensate against, so one solution is to restrict the period over which the average is collected to ensure the load is consistent. This is done by considering short trace periods in section 9.1.2, and in a real system by using a sliding window.

7.5 Summary

The load distribution mechanisms presented support both initial placement and process migration transfer mechanisms. The load metric is a three dimensional vector of primary resource utilisations, and the load is distributed through the system from a central collection agent (GLA). The prediction mechanism averages previous behaviour to provide a form of *a priori* information on jobs.

Chapter 8

Algorithms for Load Distribution

The construction of an algorithm to test the *process mix* hypotheses from chapter 3 is non trivial. Various restrictions of the load distribution mechanism, and associated costs obstruct direct implementation, and may degrade or obscure the hypothesis.

Sections 8.1 and 8.2 detail generic initial placement and migration algorithms. Section 8.3 presents the resource based distribution algorithms developed in this thesis. Each algorithm is presented with a brief description, followed by the algorithm itself and a summary of it in terms of the taxonomy from section 2.2.3. All of the algorithms are sender initiated, and no receiver may refuse a migrated or initially placed job. In the case of initial placement, the job must be executed by the receiving host, and may not be diverted elsewhere.

For brevity, the conventions from table 8.1 are adopted in the presentation of the load distribution algorithms.

	Description
T	The load threshold.
cs	Specifies the local host. (current server)
rs	The remote host. (remote server)
N	The number of hosts.
n	The number of active processes on a host.
j	The candidate job.
p	The candidate process.
$\parallel \mathbf{x} \parallel$	The norm (length) of vector $\mathbf{x}$.
$\overline{\mathbf{x}}$	The average of $\mathbf{x}$

Table 8.1: *The naming conventions used in the load distribution algorithms*

8.1 Initial Placement Algorithms

Initial placement is the distribution of jobs to hosts prior to the start of execution. Once a host has been selected and the job started, there is no way to move the job to another host. This restriction results in lower placement costs, as there is no need to transfer an address space as would be required if processes were redistributed.

The algorithms presented in this section progressively increase from random to average in both the degree of information required and the complexity of the corresponding decision.

8.1.1 Random

There is no load distribution policy simpler than random. If the load on the local host is over the load threshold, another host is chosen at random and the job started there. The random algorithm is:

$$\begin{aligned}&\textit{if } \overline{\mathbf{cs}} > T \textit{ then}\\&\quad rs = random\end{aligned}$$

Random is a stateless and non cooperating algorithm, no load information is considered or exchanged and it is aptly described as *oblivious* by Chakrabarti et al. [CRY94]. There is no restriction on which hosts are chosen as the remote execution site, and therefore there is an equal probability that the local host will be selected to execute the job. This lack of restriction is simple, even though the local host exceeds its threshold, it may still be the least loaded host. This is especially important when the threshold is low, in order to avoid running all jobs remotely.

The participation policy is embodied in the first line of the algorithm, and the location policy is entailed in the second. In summary random uses the following components:

- **Participation:** Single load threshold, no rejection.
- **Candidate Selection:** All jobs are eligible and scheduled on arrival.
- **Location Selection:** Random, no consideration is made of the remote load.

- **Transfer Mechanism:** Initial placement.
- **Load Metric:** Standard (Average of resource utilisation).
- **Load Communication:** None.

Random is intended as a simple initial placement algorithm against which to compare the worth of more complex methods. Eager et al. [ELZ86b] in an early study indicated that random provides performance close to that of complex schemes with much less effort, and Chakrabarti et al. [CRY94] found that in some situations random is within a small constant factor of optimal.

Random can be extended to test if the selected destination is below the threshold. This *probing* avoids fruitless placements to overloaded hosts, but in effect did not provide significantly better performance over random. Perhaps due to the small number of hosts in the test distributed system. For this reason, simple random is chosen to represent the very low state initial placement algorithms.

8.1.2 Least Loaded

The least loaded algorithm represents a significant increase in state over the random approach. When the load threshold is exceeded on the local host i, the host with the lowest load is selected as the destination offloading jobs.

$$\begin{array}{l} \textit{if } \overline{\mathbf{cs}}_i > T \ \textit{ then} \\ \quad rs = \textit{min}(\overline{\mathbf{cs}}_0, \overline{\mathbf{cs}}_1, \ldots, \overline{\mathbf{cs}}_N) \end{array}$$

Accuracy of Periodic Updates

The use of a periodic update with the least loaded algorithm leads to some problems with out-of-date host state information. There is no immediate feedback when a job starts executing on a host, thus the current least loaded host will continue to be selected as the least loaded host for the entire period between updates. The direct consequence of this is, all eligible jobs starting in the system will be initiated on just one host.

The situation is further aggravated when all, or a large proportion of jobs can be remotely executed, which can lead to *swamping* of the selected host, and poor system performance as shown in figure 8.1.

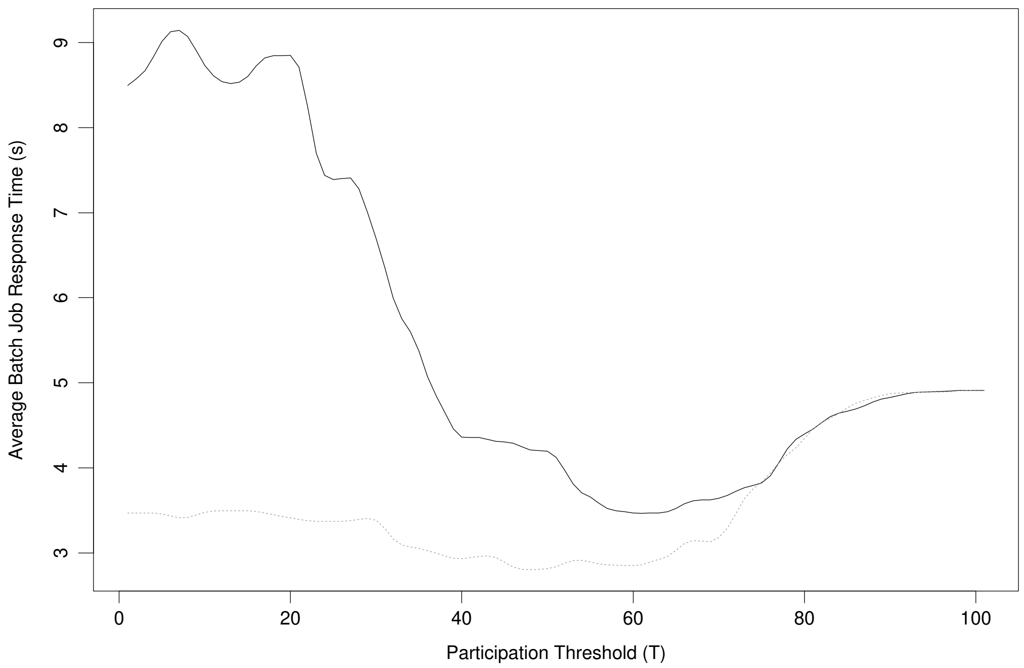


Figure 8.1: *The effect of swamping is illustrated by the solid line. The dotted line uses the same least loaded distribution algorithm, but uses the anti-swamping algorithm to estimate the load between updates.*

Anti-Swamping

There are a number of ways of minimising swamping of the least loaded host and these include:

1. Distributing fewer jobs during each interval.
2. Updating the host load information before each placement.
3. Limiting the number of placements to each host during each interval.
4. Estimating the new load after each placement.

Each of these solutions has a particular application in mind, and none are suitable in every situation. The first is too restrictive, and with a high arrival rate may not be able to redistribute sufficient load. The second is essentially polling, and can result in unwarranted degree of overhead. The third requires a large pool of hosts as in a network of workstations (NOWs). Finally, the fourth may require the load estimates to be global, especially when there is only a small set of hosts with a more than one user per host.

The testbed simulator is based on the Xterminal-Server model, in which each host services many processes concurrently. Thus estimation of the load will provide the most suitable means to determine the effect of placing a new job on a host, and will allow the current (estimated) least loaded host to be identified during the update interval.

The Anti-Swamping Algorithm

The least loaded algorithm does not have access to *a priori* information on the job that it is distributing. Therefore a reasonable approach is to treat all jobs as needing an equal portion of the hosts resources. Placing a job on a host increases the number of processes by one, and as all jobs are considered equal, the new load on the host is estimated as:

$$\overline{\mathbf{cs}} = \overline{\mathbf{cs}} \times \frac{n+1}{n}$$

A special case arises when there are no active processes, yet there is load reported from the previous utilisation period. Here the load is assumed to be the contribution of a single process that has finished, and therefore is also the contribution of the new job. Figure 8.1 illustrates the improvement in performance due to the anti-swamping technique.

Vernemmen and D'Hollander [VD91] also use estimation to provide load information between updates, but use an adaptive prediction term and a decay rate.

In summary, the components of the least loaded algorithms are:

- **Participation:** Single load threshold, no rejection.
- **Candidate Selection:** All jobs are eligible and scheduled on arrival.
- **Location Selection:** Least loaded.

- **Transfer Mechanism:** Initial placement.
- **Load Metric:** Standard.
- **Load Communication:** Standard periodic, plus estimation between updates.

Least loaded is an example of a little more intelligence in the location selection policy but still no effort to select suitable candidates.

8.1.3 Average

The previous algorithms make no effort to prevent short lived jobs from being remotely executed. This is a problem, as the time invested in setting up a remote execution, is wasted if the process terminates after a short time. This problem manifests itself primarily as a severe slowdown of short lived processes.

- **Candidate Selection :**

  The average algorithm attempts to resolve this shortcoming by restricting the jobs eligible for initial placement. Specifically, only a job j that has in the past, executed for longer than the time required to remotely initiate it, is eligible:

  $$j.\tau > f \times \eta$$

  where,

  - τ is the expected lifetime of j,
  - η is the cost of placing it,
  - and f is a constant factor, arbitrarily set to 1.

  The constant factor f is a parameter for tuning the candidate selection policy. As no tuning is carried out in this thesis, the value for f is selected to give a worst case response time of 2τ for remote execution (of course it could be worse than 2τ, but that would not be the fault of the candidate selection policy).

- **Location Selection :**

  There is no particular constraint on the location policy, and both random and least loaded are possible means for locating a destination host. However figure 8.2 demonstrates random is not a suitable location policy when combined

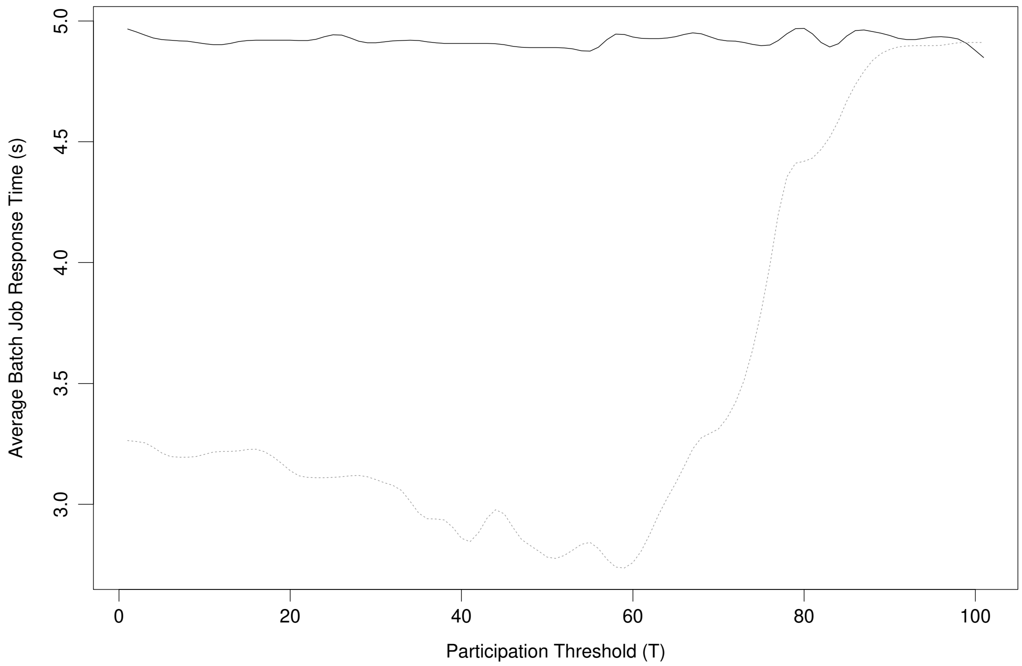


Figure 8.2: *Average with random (solid line) and least loaded (dotted line) location policies.*

with average based candidate selection. This is possibly due to having too few jobs for a random policy to distribute after the candidate selection phase.

Thus the complete average algorithm is:

$$\begin{array}{l} \textit{if } \overline{\mathbf{cs}} > T \\ \quad \textit{if } j.\tau > f \times \eta \\ \quad\quad rs = \textit{least\_loaded} \end{array}$$

where, if the job is expected to execute for longer than $f \times \eta$ it is executed on the least loaded host.

Thus the average load distribution algorithm has:

- **Participation:** Single threshold, no rejection.
- **Candidate Selection:** Only jobs with an expected duration greater than the placement delay are eligible, and are scheduled on arrival.

- **Location Selection:** Least loaded.
- **Transfer Mechanism:** Initial Placement.
- **Load Metric:** Standard.
- **Load Communication:** Standard periodic, plus estimation used by the least loaded location algorithm.

The intention of this algorithm is to gauge the effect of using a small amount of historical information for load distribution. As only longer jobs are eligible for distribution, the amount of movable workload is correspondingly less. This however should not be a problem as smaller jobs will no longer suffer from a proportionally larger distribution overhead.

8.2 Migration Algorithms

Migration offers more flexibility than initial placement, as processes may be moved at anytime during their execution. This tends to result in process migration policies being more complex than their initial placement counterparts, as data from the current execution is available for selecting candidate processes.

This section presents two process migration algorithms, the first is from the existing literature and is remarkable for its use of minimal state. The second is created specifically for comparing with the *process mix* based migration algorithms.

8.2.1 Harchol-Balter and Downey

Harchol-Balter and Downey [HBD95] resurrect early work by Leland and Ott[LO86] that shows process lifetime distributions exhibit a long thick tail rather than the exponential or hyper exponential assumptions used in most later work. In particular, Harchol-Balter and Downey present the following *rules of thumb*[1] *(page 2):*

1. The probability that a process with age of 1 second uses T seconds of CPU time is about $\frac{1}{T}$.

[1] These rules of thumb have the interesting implication that the mean of the distribution is infinite.

2. The probability that a process with age T seconds uses an additional T seconds of cpu time is about $\frac{1}{2}$.

These observations form a basis for the candidate selection policy, which only uses current runtime information. From this point I will refer to the migration algorithm from Harchol-Balter and Downey as hbd-mig.

Location Selection policy

Hbd-mig is a periodically initiated load levelling algorithm. At every periodic state update, a single migration may occur between the most (source) and least (destination) loaded hosts in the system, which are selected using the following algorithm:

$$
\begin{aligned}
\textit{source} &= \textit{max}(\overline{\mathbf{cs}}_0, \overline{\mathbf{cs}}_1, \ldots, \overline{\mathbf{cs}}_N) \\
\textit{destination} &= \textit{min}(\overline{\mathbf{cs}}_0, \overline{\mathbf{cs}}_1, \ldots, \overline{\mathbf{cs}}_N)
\end{aligned}
$$

Thus the source from which a process is transferred is the host with the maximum load, and the destination is the host with the minimum load.

Candidate Selection policy

Once the source and destination have been located, the candidate selection policy attempts to find a process on the source host that will receive better service at the destination host as well as executing for a sufficient length of time, compared to the cost of moving it.

This identifies two components in the candidate selection policy:

- **Eligibility:** The eligibility of a process is defined in terms of *fairness*, see section 2.3.3. Process p_i is only eligible for migration if it will experience better service on the destination host:

$$\forall p_i,\ p_i \in E \quad \textit{iff} \quad p_i.\delta > \frac{p_i.\gamma}{n - (m+1)}$$

  where,

- E = the set of eligible processes,
- n is the number of processes on the source host,
- m is the number of processes on the destination host,
- the potential migrant is represented by adding one to m,
- δ is the current age of process p_i and
- γ is the estimated cost of migrating process p_i.

This metric is applied to all processes executing on the source host to form the set of eligible processes, from which the actual candidate is selected.

- **Selection:** Harchol-Balter and Downey consider that the best candidate in the set of eligible processes E, is the candidate with the greatest current age over the cost of migration:

$$max(\frac{p_0.\delta}{p_0.\gamma}, \frac{p_1.\delta}{p_1.\gamma}, \ldots, \frac{p_e.\delta}{p_e.\gamma})$$

where, $e = |E|$ (the cardinality) and all other symbols are as defined previously.

This policy is based directly on the *rules of thumb*, where the current age of a process is expected to double with a probability of a half.

A load threshold participation policy is added to the original hbd-mig algorithm to allow full comparison with the other algorithms. The critical features of this algorithm are:

- **Participation:** Single threshold, no rejection.
- **Candidate Selection:** Expectation of better service plus maximum process age to migration cost ratio. A process may be migrated at any time during its execution.
- **Location Selection:** Periodic global selection of least and most loaded.
- **Transfer Mechanism:** Process Migration.
- **Load Metric:** Standard.
- **Load Communication:** Standard periodic.

In summary, hbd-mig is a simple load distribution system which uses little state. This makes it an ideal basis for comparison with the resource based migration algorithms which require a great deal more state.

8.2.2 Least Loaded Migration

The purpose of the least loaded migration scheme (ll-mig) is to implement a location policy similar to the following resource based migration algorithms, yet does not attempt to adjust the process mix. Thus the sole purpose of this algorithm is to provide a baseline with which the later algorithms can be meaningfully compared.

- **Location Selection Policy :**

  The location selection policy is identical to that of hbd-mig:

$$\begin{aligned} source &= max(\overline{\mathbf{cs}}_0, \overline{\mathbf{cs}}_1, \ldots, \overline{\mathbf{cs}}_N) \\ destination &= min(\overline{\mathbf{cs}}_0, \overline{\mathbf{cs}}_1, \ldots, \overline{\mathbf{cs}}_N) \end{aligned}$$

- **Candidate Selection Policy:**

  The candidate selection policy has the simple objective of redistributing the largest amount of load from the source host.

  Eligibility: To ensure that this distribution isn't fruitless, it uses a candidate eligibility based on the average candidate selection policy:

$$\forall p_i,\ p_i \in E \quad iff \quad p_i.\tau > f \times p_i.\gamma$$

  where,

  - E = the set of eligible processes,
  - τ is the expected lifetime of process p_i,,
  - γ is the cost of migrating process p_i,
  - and f is a constant factor that is arbitrarily 1.

  Candidate Selection:

  Given that the eligible candidates have a reasonable portion of lifetime remaining, the 'best' process is selected by the load it imposes on the system. This load is determined from the average utilisation that the process imposes on each of the primary resources:

$$max(\overline{\mathbf{p}}_0, \overline{\mathbf{p}}_1, \ldots, \overline{\mathbf{p}}_e)$$

  where $e = |E|$.

Therefore ll-mig can be summarised as:

- **Participation:** Single threshold, no rejection.
- **Candidate Selection:** Only processes with an expected duration greater than the placement delay are eligible. Selects the largest user of resources from the eligible set.
- **Location Selection:** Periodic global selection of least and most loaded.
- **Transfer Mechanism:** Process Migration.
- **Load Metric:** Standard.
- **Load Communication:** Standard periodic.

8.3 The Resource Based Algorithms

The Resource based algorithms described in this chapter seek to maintain the *process mix* hypothesis, which is:

Process Mix: The distribution of a balanced mix of CPU, memory and IO bound processes to each host will lead to the more effective use of resources and delay the onset of bottlenecking.

Difficulties arise from restrictions imposed by both the real time constraints of a computer system, and the implementation practicalities of a load distribution mechanism. Either of these restrictions can degrade or obscure the effect of load distribution, and make the algorithm useless for evaluating the hypothesis.

To truly maintain the *process mix* of a system, would require an instantaneous and free redistribution of all processes in the system. The problem can be simplified to one of *reducing the imbalance in demand for individual resources within a host and between the hosts*, which is a much more realistic proposition.

This approximation is equivalent to the *process mix* hypothesis, under the restriction that only one process or job may be distributed at any one time. This equivalence arises in the following way:

- The demand for resources on a host is equal to the sum of the demands from all processes on that host.
- The best mix occurs when the resource demands of the candidate process are complementary with the sum total of the demands due to all other processes on the host.

The remainder of this section is devoted to describing the implementation details of the resource based algorithms. The common metric and fitting techniques are developed in sections 8.3.1 to 8.3.2 and the resulting initial placement and process migration algorithms are presented in sections 8.3.3 and 8.3.4 respectively.

8.3.1 Resource Balancing Metric

The n resources a host can provide to service processes can be considered to form an n dimensional space. In particular, if a host provides CPU, file IO and memory, then these form a 3 dimensional space, in which a single point locates the current state of the host.

If the values of resource use are constrained to the range of $0 \rightarrow 1$, then a completely unloaded host would locate at the origin $< 0, 0, 0 >$ and a completely loaded host would be located on the opposite corner at $< 1, 1, 1 >$. This space is illustrated in figure 8.3.

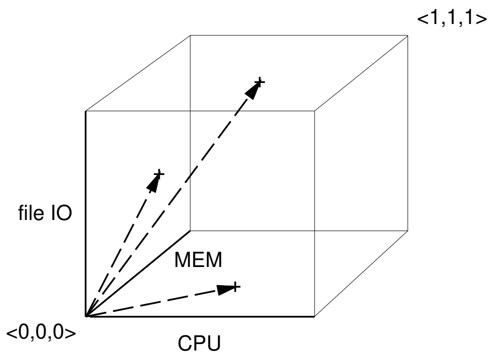


Figure 8.3: *The 3 dimensional space used to describe current host state, and the three points indicate potential host loads.*

Examination of this space identifies two separate components of the load on a host:

- The volume enclosed by perpendicular lines from the point to each of the axes (the Euclidean measure) represents the current load on the host, or in other words, to what extent the host's resources are allocated.

- The variance of the separate components in the vector to the point determines how balanced the allocation of resource is.

To enable hosts to be ranked for placement suitability, a single metric is required to combine these two load components. One metric is to measure the Euclidean distance between the point and the origin, or in slightly different terms, the *norm* (length) of the vector from the origin to the point. This is shown graphically with two resource dimensions in Figure 8.4 , where a well balanced host **b**[2], is ranked equal to a less balanced, yet more lightly loaded[3] host **a**.

This tradeoff between balance and load works due the property that a vector on the diagonal always encloses a greater volume than one of equal length offset to either side.

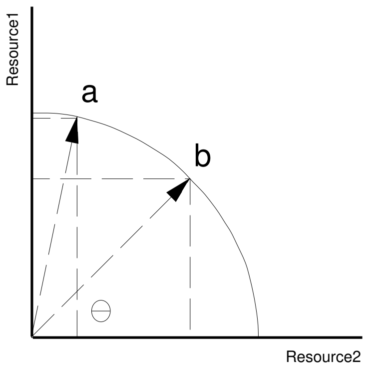


Figure 8.4: *The different volumes enclosed by two vectors with the same norm (length).*

The area (load) alone is not suitable for ranking hosts **a** and **b**, as this ranking would always prefer **a** to **b**, and therefore does not account for the fact that resource 1 is closer to saturation. The variance of the host resources is also not sufficient on its own, as it provides no means of detecting an overloaded host (just an unbalanced one).

Thus if two hosts are to be compared, the norm provides a simple means of ranking them when both balance and load are significant.

8.3.2 Resource Fitting

Section 8.3.1 detailed how a 3 dimensional vector can be used to describe the current state of the host, and that the norm of this vector can be used to rank host loads. Jobs

[2]Balance in this case is indicated by $|45° - \theta| = 0$

[3]Encloses less area, denoted with the dashed lines

can also be described and ranked using these same resource dimensions, and if they are expressed in the same units as the host, the job can be added to and subtracted from the host vector. The unit that most naturally expresses both the resource availability on a host, and resource demand from a process is utilisation. This enables jobs to be trial fitted to hosts in order to select the one that is most suitable.

There are three standard methods of fitting resource requests to resources available that can be adapted from the domain of physical memory allocation.

- **First Fit:** Select the first host that has sufficient resources to meet the request.
- **Best Fit:** Select the host with the smallest remaining volume that most closely fulfils the request (minimises leftover).
- **Worst Fit:** Select the host that provides the largest volume of resources that fulfils the request (maximises leftover).

There are a number of differences in the application of the physical memory fitting algorithms when used for allocating host resources. Firstly, physical memory is a finite resource and cannot be over allocated, whereas host resource limits are softer and can withstand a degree of overallocation yet still satisfy the request. Secondly, the ideal situation is to completely allocate a segment of physical memory leaving other segments completely free (ignoring fragmentation), this is opposite to the desired effect of host resource allocation, where it is desirable to have all hosts with the same degree of allocated resources (in other words, as condensed versus as spread, as possible).

At this point it is useful to describe the fitting methods in more detail before discussing the actual load distribution algorithms.

First Fit

First fit consists of two functions, the selection and the fitting. These two phases are often intertwined as in the fastest response of V [TLC85], described in section 2.3.5. The essence of this system is that the first host to reply to a multicast request, by its very reply implies that it has sufficient resources to service the request. The first fit implementation described in this section has a more explicit fitting phase.

For a resource based algorithm, first fit can be implemented by adding the components of the job and host vectors, and comparing the results to the maximum value defined for each component. If none of the maximums are exceeded, the job is considered to fit, and is executed on that host.

Best Fit

The best fit from Cena et al. [CCG95] detailed in section 4.2.5, attempted a best fit based on specified job resource requirements. The goal of this scheme is to ensure the highest degree of remaining allocable resources in a heterogeneous system. It did not attempt to mix the processes on a host to ensure complementary behaviour.

Best fit requires that the resource request is most closely satisfied, which has the direct effect of loading up already loaded hosts, in order to leave the larger 'more allocable' blocks free. This is contrary to the basic aim of load distribution, and for this reason alone, best fit is not pursued any further in this thesis.

Worst Fit

Worst fit operates in the opposite way to best fit. Instead of choosing the closest fit, worst fit selects the host that results in the largest leftover. This is implemented by selecting the host for which the addition of the job results in the smallest norm as shown graphically by figure 8.5.

This fitting technique is used to approximate the maintenance of the *process mix* on each host.

Summary

From this discussion on resource fitting it is clear that there are two fitting techniques suitable for load distribution, first and worst fit. Finding the worst fit approximates the maintenance of a balanced *process mix*, and is a reasonable approach for both initial placement and process migration. In contrast, first fit does not maintain the *process mix* nor is it a natural location policy for process migration, and as such, the only first fit algorithm implemented uses initial placement.

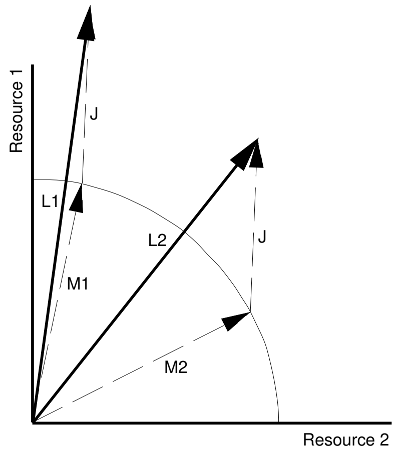


Figure 8.5: *In this 2D space formed by resources 1 and 2, there are two hosts M1 and M2, of equal load. M1 is heavily utilising resource 1 and host 2 is heavily utilising resource 2. Job j, which is resource 1 bound, is trial added to both hosts. The resulting length of L2 is less than L1, and therefore the job is allocated to M2.*

As the fitting techniques are borrowed from physical memory allocation, the names are somewhat unsuitable. I will however continue to refer to each technique by its original name to avoid confusion.

8.3.3 Resource Based Initial Placement

Resource based initial placement needs the resource usage of a job in order to fit it to the most suitable host. Since initial placement requires that jobs are scheduled before execution, any resource requirements must be provided by estimates based on historical behaviour.

First Fit

First fit initial placement (ff-ip) is a direct implementation of section 8.3.2.

- **Candidate Selection :**

  The candidate selection (or to be more accurate, the candidate eligibility) is based on the availability of predicted job characteristics, where the expected lifetime of a job j must be greater than the cost of placing it:

$$j.\tau > f \times \eta$$

where,

- τ is the expected lifetime,
- η is the cost of placing it,
- and f is a constant factor that is arbitrarily set to 1.

This is the same candidate selection policy that is used by average.

- **Location Selection :**

  The location selects the first host that can meet the resource requirements of an incoming job $\mathbf{j}$[4]:

$$\mathbf{cs}_R + \mathbf{j}_R < lim_R, \qquad \forall R = \{CPU,\ Memory\ and\ IO\}$$

  where lim_R is the maximum amount of resource R that the host cs can supply.

The implementation of the first fit algorithm uses the periodic global state information provided by the periodic updates. First fitting occurs in a cyclic fashion starting at the host following the last successful fit. The new load is estimated after each placement and this is described on page 126.

Thus ff-ip can be summarised by:

- **Participation:** Single threshold, no rejection.
- **Candidate Selection:** Only jobs with an expected duration greater than the placement delay are eligible, scheduled on arrival.
- **Location Selection:** Cyclic first fit.
- **Transfer Mechanism:** Initial placement.
- **Load Metric:** Normal of 3 space point.
- **Load Communication:** Periodic with estimation.

First fit uses the standard prediction mechanism to provide job resource requirements prior to execution.

[4] Algorithm terms are defined on page 107

Worst Fit

Worst fit initial placement (wf-ip) seeks to locate the host i that results in the shortest norm of the sum of the host, $\mathbf{cs}_i$, and job, $\mathbf{j}$, vectors.

- **Candidate Selection :**

  The candidate selection policy is the same as that used by average and first fit, with:

  $$j.\tau > f \times \eta$$

- **Location Selection :**

  Given the job is eligible, the fit of the job is computed for each host, and the destination host is the:

  $$min(\parallel \mathbf{cs}_0 + \mathbf{j} \parallel, \parallel \mathbf{cs}_1 + \mathbf{j} \parallel, \ldots, \parallel \mathbf{cs}_N + \mathbf{j} \parallel)$$

  No specific favour is given to the local host, other than in the computation of the job's contribution to IO utilisation, where the utilisation estimate uses the cheaper local IO cost.

In summary, wf-ip consists of the following:

- **Participation:** Single threshold, no rejection.
- **Candidate Selection:** Only jobs with an expected duration greater than the placement delay are eligible, scheduled on arrival.
- **Location Selection:** Minimum norm of job and host vector sum.
- **Transfer Mechanism:** Initial placement.
- **Load Metric:** Normal of 3 space point.
- **Load Communication:** Periodic with load estimation between updates.

Closeness

As an alternative to using a load threshold, the location policy could be extended to consider the difference between the local and selected host's norms. If they are within some closeness factor, say a percentage difference, then the local host should be selected.

Load Estimation

The load distribution mechanism supplies periodic global state updates to all of the load distribution policies. This presents a problem for the two initial placement fitting algorithms, as no immediate feedback to a job placement is provided. Therefore fittings subsequent to the first in each period are made with data that is known to be out of date, and similar problems to those encountered by the least loaded initial placement algorithm may arise.

One advantage of using predicted resource requirements is that the same trial fitting technique that is used to select the destination host, can be used to provide an estimate of the load after job placement. Thus the result of the fitting computation is used to directly update the load value stored for the destination host.

No explicit action is taken to correct the load estimates, even when a short job (less than the remaining period) is scheduled, as I have assumed that the periodic state update is sufficient to correct for this.

8.3.4 Worst Fit Migration

Process migration presents greater choice in the implementation of resource balancing than the equivalent initial placement scheme. The primary reason being, that any process being executed in the system may be selected at any time, compared to initial placement, where only an arriving job can be considered for remote execution.

Therefore two process migration approaches are presented for worst fit, rather than only one as with initial placement.

- **Fairest Candidate :**

    1. Find the most and least loaded hosts.

2. Select the candidate that minimises the load on both the most and least loaded hosts.

- **Highest Contributing Candidate :**

1. Locate the most loaded host, and select the candidate which minimises the load.
2. The candidate is then worst fitted to all hosts in the system.

Whilst on the surface these two approaches appear similar, the first chooses the best candidate to suit the source and destination hosts, while the second chooses the destination to best suit the candidate.

The 'better' alternative will be determined experimentally in chapter 9.

As with hbd-mig, all of the resource based process migration algorithms are periodic. They require global host state information, and as a consequence their activity follows the global state update. There is at most one migration in the system per global state update, and therefore both approaches must be considered as load levelling rather than load balancing. As migration is initiated after each global state update, any migration decisions are made with the most recently available data, and there is no need to estimate the load between updates.

Candidate Eligibility

Both migration approaches share the same eligibility test in the candidate selection policy, which is incidentally the same as for ll-mig:

$$\forall p_i,\ p_i \in E \mathit{iff}\, p_i.\tau > f \times p_i.\gamma$$

where,

- $E =$ the set of eligible processes,
- τ is the expected process lifetime,
- γ is the cost of migrating it,
- and f is a constant factor that is arbitrarily 1.

and $e = |E|$.

Fairest Candidate

This algorithm consists of two major parts, the location selection and the candidate selection:

- **Location Selection Policy :**

  This is a sender initiated policy, where on receiving the global state update, the machine which is most loaded identifies itself, **cs**, and also identifies the least loaded host, **rs**. The metric is the based on the norm:

  $$cs = max(\| \mathbf{cs}_0 \|, \| \mathbf{cs}_1 \|, \ldots, \| \mathbf{cs}_N \|)$$
  $$rs = max(\| \mathbf{rs}_0 \|, \| \mathbf{rs}_1 \|, \ldots, \| \mathbf{rs}_N \|)$$

  A tie is resolved by selecting the host with the lowest id. With the source and destination locations decided, the next step is to identify the candidate.

- **Candidate Selection Policy :**

  This algorithm is based on the *process mix* hypothesis, so there is no sense in selecting the largest process from the source host, and moving it to the destination without considering the effect on the process mix of the destination.

  This being the case, the ideal candidate is the one that would decrease the norm of the source host by the maximum amount and increase the norm of the destination host by the minimum amount. These contrary demands can be compromised by selecting the candidate process p with the minimum:

  $$\| \mathbf{cs} - \mathbf{p} \| + \| \mathbf{rs} + \mathbf{p} \|$$

Fairest worst fit is summarised as:

- **Participation:** Single load threshold, no rejection.
- **Candidate Selection:** $\| \mathbf{cs} - \mathbf{p} \| + \| \mathbf{rs} + \mathbf{p} \|$
- **Location Selection:** Least loaded host.

- **Transfer Mechanism:** Process Migration.
- **Load Metric:** Norm of the load vector.
- **Load Communication:** Standard periodic.

Highest Contributing Candidate

This differs from the previous algorithm in that the candidate process is selected before the destination host, and therefore there are no conflicting requirements to meet in the selection of the candidate process.

- **Location Selection Policy (Source only) :**

  This is a sender initiated policy, where on receiving the global state update, the machine which is most loaded identifies itself, with the following metric based on the norm:

  $$max(\| \mathbf{cs}_0 \|, \| \mathbf{cs}_1 \|, \ldots, \| \mathbf{cs}_N \|)$$

  Any ties are again resolved by selecting the host with the lowest id. With the source location decided, the next step is to identify the candidate.

- **Candidate Selection Policy :**

  There is more flexibility in the selection of candidates for this algorithm, as there is no destination host yet specified to restrict the choice of candidates. Thus there are four different candidate ranking formulae presented:

  1. The simplest ranking is to use the norm of the load imposed on the host by process p_i:

     $$max(\| \mathbf{p_0} \|, \| \mathbf{p_1} \|, \ldots, \| \mathbf{p_e} \|)$$

  2. This can be extended by considering the expected remaining lifetime, τ, of the candidate process p_i:

     $$max(\| \mathbf{p_0} \| \times\tau_0, \| \mathbf{p_1} \|_1, \ldots, \| \mathbf{p_e} \| \times\tau_e)$$

3. A further extension can consider this relative to the estimated cost γ of migration for process p_i:

$$max(\| \mathbf{p_0} \| \times \frac{\tau_0}{\gamma_0}, \| \mathbf{p_1} \| \times \frac{\tau_1}{\gamma_1}, \ldots, \| \mathbf{p_n} \| \times \frac{\tau_e}{\gamma_e})$$

4. The resources utilised by a young process may not be representative of its normal behaviour due to a short sample period. Unfortunately there is a better pay back the earlier in its execution that a good candidate process is migrated. One way to ensure that young processes are represented properly and selected fairly, is to use historical resource demands, as in wf-ip and Ju et al. [JXY95]. The major difference is that the predicted data is used to form a *worst case* assessment of the processes behaviour (i.e., the worst behaviour from current and predicted resource usage), and this estimate is the used to select the candidate.

$\mathbf{p}_i = \| < max(cpu_{real}, cpu_{est}), max(mem_{real}, mem_{est}), max(io_{real}, io_{est}) > \|$
$max(\mathbf{p}_0, \mathbf{p}_1, .., \mathbf{p}_e)$

- **Destination Selection Policy :**

  The destination selection policy must now locate the host which is most suitable for the candidate $\mathbf{p}$. This is done in exactly the same manner as wf-ip:

$$min(\| \mathbf{cs}_0 + \mathbf{p} \|, \| \mathbf{cs}_1 + \mathbf{p} \|, \ldots, \| \mathbf{cs}_N + \mathbf{p} \|)$$

Highest contributing worst fit is summarised as:

- **Participation:** Single load threshold, no rejection.
- **Candidate Selection:** Any of policies 1 through 4 from page 129
- **Location Selection:** Worst fit host.
- **Transfer Mechanism:** Process Migration.
- **Load Metric:** Norm of the load vector.
- **Load Communication:** Standard periodic.

8.4 Summary

This section presented eight algorithms for load distribution, these are summarised in table 8.2.

	Component					
	Participation	Candidate	Location	Transfer	Metric	Comm.
Random	Threshold	All	Random	IP	Av. Util.	None
Least Loaded	Threshold	All	LL	IP	Av. Util.	Periodic
Average	Threshold	$\tau > \eta$	LL	IP	Av. Util.	Periodic
hbd-mig	Threshold	$p.\delta > \frac{p.\gamma}{n-(m+1)}$, $\max \frac{p.\delta}{p.\gamma}$	LL	MIG	Av. Util.	Periodic
ll-mig	Threshold	$p.\tau > p.\gamma$, max $\overline{\mathbf{p}}$	LL	MIG	Av. Util.	Periodic
ff-ip	Threshold	$\tau > \eta$	FF	IP	Axis Limit	Periodic
wf-ip	Threshold	$\tau > \eta$	WF	IP	Norm	Periodic
wf-mig	Threshold	Various	WF	MIG	Norm	Periodic

Table 8.2: *A summary of all eight load distribution algorithms.*

Each of these algorithms explores some difference in policy, culminating in the worst fit algorithms. The worst fit algorithms are based on an approximation to the *process mix* hypothesis, and the remaining algorithms are to test if the hypothesis holds by comparison.

For the purposes of evaluating the *process mix* derived algorithms, the closest possible initial placement technique to wf-ip without resource fitting is average. The other fitting algorithm ff-ip, is less close, but as it uses prediction it provides another useful basis for comparison. For process migration, the closest possible to wf-mig, without resource fitting is ll-mig. The hbd-mig algorithm provides a reference to current conventional approaches.

Chapter 9

Experimental Work

Chapter 3 introduced the principle of maintaining a balanced *process mix* on each host to improve the performance of the overall system. This chapter presents experiments which compare the performance of algorithms that balance multiple resources against those that don't. These algorithms are evaluated using the testbed described in chapters 5 through 7, and test the validity of the *process mix* hypothesis.

The first section of this chapter is concerned with the selection of suitable traces for driving the testbed. The second section deals with the collection and measurement of the testbed output, as well as the selection of the experimental parameters. The third section presents and analyses the experimental results, and the final section summarises and concludes the findings of this chapter. The major finding is that:

> *The maintenance of a balanced process mix, consistently reduces the average response time of jobs within a system, indicating better resource availability.*

9.1 Trace Selection

The composition of the workload can have a significant effect on the performance of a load distribution algorithm. This poses a problem for the evaluation of load distribution as even a real workload trace may obscure the true nature of an algorithm if the period is poorly chosen. This section describes the selection of suitable test periods from the workload trace.

9.1.1 Features of Interest

Casavant and Kuhl summarised the underlying philosophy of load distribution as "...being fair to the systems hardware resources is good for the users of that system". With this in mind, a good test is to determine if all resources are being treated fairly, especially when the demand for a single resource is disproportionate or biased.

Periods of light workload generally offer little insight into the value of load distribution, as users will obtain satisfactory performance from almost any algorithm. A heavy workload is of more interest, with greater contention for resources and a corresponding increase in waiting time.

In summary, two workload attributes suitable for evaluating load distribution are:

- **Biased Periods:** A workload biased towards a particular resource will test the fairness of each algorithm towards individual system resources.
- **Heavy Periods:** A heavy workload will test the algorithms ability to reduce the process waiting time, when all resources are in demand.

The best test period will combine a high workload and a bias towards a particular resource, or subset of resources.

9.1.2 Trace Period Length

The length of the trace periods is fixed at one hour, giving 168[1] potential sample periods. This period is a natural choice for several reasons:

- An hour is short enough to give a reasonable simulation length.
- The period is sufficiently short to avoid the averaging effect that would occur over longer periods, yet also long enough to exhibit a variety of activity.
- The traces exhibited variable behaviour from hour to hour, possibly coinciding with the start and finish of the undergraduate laboratories and lectures.

[1] Traces for the entire month of May 1995 were recorded, however the most interesting part of the trace was the last week of term. Thus all samples are drawn from this period, giving a total of 168 one hour periods.

The Test Set

A biased period can be detected using arbitrary thresholds based on the total demand for each resource within the hour. However, a threshold such as this is difficult to quantify, as it will vary with the workload composition in each hour. A better approach is to rank each hour by the total demand for each primary resource, and to assume that the greater the difference between the individual resource rankings of a period, the greater the degree of resource bias.

The primary resources simulated are CPU, memory and file IO, giving the six possible bias combinations enumerated in table 9.1. In addition, two periods with high and medium loads and no bias, complete the range of test workloads.

		Ranking			**Arrivals**	
Trace	Bias Type	CPU	Memory	file IO	Ranking	Arrivals
1	CPU	3	-	-	51	2703
2	memory	-	11	-	5	6477
3	file IO	-	-	6	91	1701
4	CPU and memory	6	12	-	7	5914
5	CPU and file IO	17	-	12	40	3095
6	Memory and file IO	-	6	18	15	4829
7	Non biased, heavy load	1	1	2	1	8584
8	Non biased, medium load	40	29	39	28	3885

Table 9.1: *The rankings of the selected trace periods. A dash indicates a ranking lower than 30^{th}, which I have omitted for clarity.*

These rankings indicate a lack of correspondence between the arrival rate and resource consumption. Consider for example, trace 3, which exhibits the sixth highest IO consumption, yet has an arrival rate ranking of 91^{st} placing it in the lower half of all 168 one hour periods. However, it is satisfying to note that the period with the highest cumulative ranking of its resources, is also the period with the highest arrival rate.

9.2 Gathering measurements

Chapter 5 found that load distribution is measurable by observing the side-effects of its operation. This section firstly presents the response time as the metric with which the performance of each load distribution algorithm is measured. Secondly, it states how the tests need to be performed to ensure that the results are valid, and lastly describes the input parameters and output values of the testbed.

9.2.1 A Performance Metric

Choosing a suitable metric is as critical as the selection of the workload. A poor metric can fail to provide useful information, or worse, provide misleading information on which faulty conclusions may be drawn. An example of a poor metric is the use of the number of processes on each host for determining the system balance, a metric which Martin [Mar94] found to be totally unrelated to the amount of work actually performed on each host.

In contrast, the *response time*[2] of *batch*[3] processes, is an excellent performance metric, as it includes all delays and service times that a process encounters during its execution. Any increase in contention for resources is directly visible as an increase in the response time.

A batch process is more suitable for evaluating load distribution than an *externally delayed* process, where only a small proportion of the time the process spends in the system can be affected by an improvement in system performance. In contrast, the response time of a batch process is only dependent on the availability of resources in the system, and will respond directly to any change in resource availability. Any change in the residency times of the externally delayed processes, due to a change in system performance, will effect the batch jobs. Thus the effect of the service times received by the externally delayed processes, are not ignored. This gives a clearer indication of any change in the overall performance of the system.

9.2.2 Fairness

A comparative study supports a hypothesis by demonstrating that an algorithm, using the principle embodied in the hypothesis, results in better performance than an otherwise equivalent algorithm which does not. The emphasis is on 'otherwise equivalent', which although obvious, has not been adhered to in other studies, of which

[2]The response time of a process is the total time it spent in the system, from initiation to termination.
[3]Defined on page 79

Goswami et al. [GDI93] is an example:

> Goswami et al. compare the performance of their predictive load distribution algorithm against CENTEX [Zho87], but provide their algorithm with more load information than CENTEX. In particular, CENTEX is restricted to a load metric based only on the length of the CPU queue, while the predictive algorithm uses both CPU and IO resource information. This difference may bias the results, as Ferrari and Zhou [FZ87] found that better performance is gained with a load metric that uses more than one primary resource.
>
> Therefore it is difficult to say whether the observed improvement over CENTEX is due to the use of prediction as claimed, or the use of a load metric with more information. A fairer test would include IO information in the load metric for CENTEX.

This form of experimental bias is avoided in the testbed simulator by using exactly the same load distribution mechanisms for all algorithms, including (where applicable), the transfer mechanism, load metric, prediction mechanism, and communication mechanism. Each algorithm is also tested with the same input parameters (see section 9.2.3), and workload traces.

9.2.3 Test Parameters

There are four main variables in the testbed: the algorithm, the participation threshold, the percentage of mobile jobs, and the fixed transfer cost (figure 9.1). Thus each algorithm has a three dimensional input surface, and when combined with the eight trace periods a large experimental space is formed.

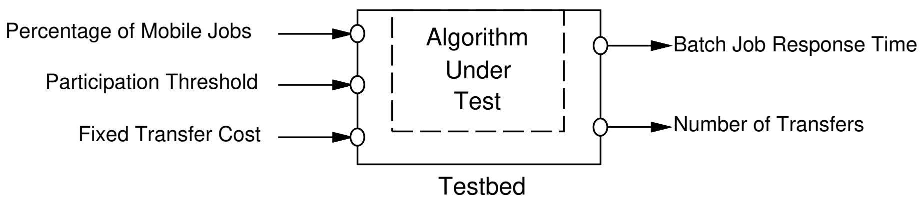


Figure 9.1: *The testbed, the input parameters and output values.*

Input Parameters

There are two types of input parameter: the *environmental* parameters, which define the environment in which an algorithm operates, specifically, the mobility of the workload and the fixed transfer cost; and the *algorithmic* parameters, such as the participation threshold, which controls how an algorithm reacts to different workload levels.

- **Workload Mobility:** The amount of workload that can be remotely executed or migrated, varies from system to system and time to time. It is unrealistic to consider that the entire workload is mobile at all times, and equally uninformative to select an arbitrary value of say 70%. Varying the degree of mobility from 0% to 100% (or in out case, the mobility factor from $0 \rightarrow 1$) will test the sensitivity of each algorithm over all possible values, a useful point on which to compare different load distribution algorithms.
- **Fixed Transfer Cost:** The fixed component of the transfer cost may also vary, often in reaction to changes in the system load. It is also unrealistic to arbitrarily fix this cost, and the algorithms response to the variation of this cost is another useful basis on which to make comparisons. The range of fixed costs considered is from from fifty milliseconds to one second, to reflect very high and slow transfer rates. Any increase in the fixed transfer cost does not alter the cost of migrating a processes address space.
- **Participation Threshold:** The participation threshold controls the point at which a host may begin to offload surplus workload. The threshold used for all algorithms is based on a percentage of the host's average primary resource utilisation. The total range of the threshold value is therefore 0%, when load distribution is attempted at all times, to 100%, when there is no load distribution at all.

Output Values

Of the statistics reported by the testbed, the two outputs of direct interest are:

- **Response Time:**[4]

[4]From now on *batch job response time* will be abbreviated to *response time*.

This load metric can be used in several ways, of which the simplest is to take the average time for all processes over all hosts in the system. A lower average response time for a set of input parameters indicates better overall service was received from the system as a whole, and as such, provides a useful measure of performance.

While there will always be variation among processes, a lower standard deviation indicates how evenly the systems resources are shared between processes.

- **Number of Transfers:** The number of remote executions or migrations performed by an algorithm under various environmental parameters, indicates how the algorithms candidate selection policy reacts to different system situations.

9.3 Results and Analysis

All of the experimental results are presented in appendices A through D. The importance of these results lie in the overall reaction of the algorithms to a variety of workloads and test parameters. Hence no results are brought forward into this chapter, but rather references are made to individual and ranges of results in the appendices.

Figure 9.2 is the key to the results present in appendices A through D, and shows the parameters selected to evaluate each of the algorithms. Each appendix represents a different set of experiments, testing the various algorithms relating to the proof of each hypothesis. Within an appendix, there are eight subsets of graphs, corresponding to the trace periods from table 9.1. Each of these subsets tests the algorithm against workload mobility and the fixed transfer cost. Ranges are specified in the form A.*.2, which denotes graph 2 for all test sets in appendix A.

The appendices represent only a fraction of the experimental space described in section 9.2.3. In particular, each graph has only one variable parameter, requiring the remaining parameters to be fixed. Table 9.2 lists the fixed parameters for each graph.

The fixing of these parameters deserves explanation, especially when different values could produce different results.

- **Fixing Mobility:** The question for graphs *.*.4 is whether selecting a different workload mobility would affect the gradient (the significant result) as the fixed

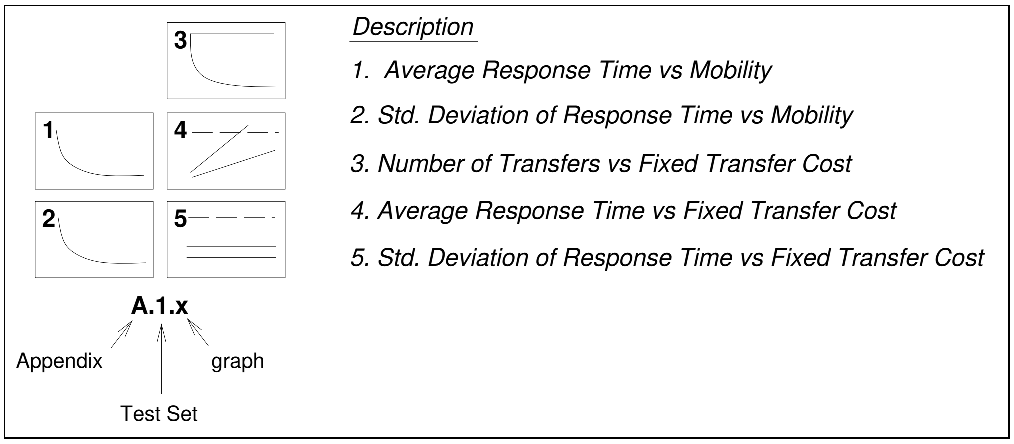


Figure 9.2: *The layout of the results in appendices A through D.*

	fixed parameter		
Variable Parameter	Mobility	Threshold	Fixed Cost
Mobility	-	0	330 ms
Fixed Cost	1	0	-

Table 9.2: *The fixed input parameters.*

transfer cost is varied. Figure 9.3 indicates that the gradient remains constant over a significant range of mobility values, and therefore fixing this parameter should not significantly alter the results for any of *.*.3, *.*.4 and *.*.5.

- **Fixed Transfer Cost:** The transfer cost is fixed for graphs *.*.2 and *.*.3. From figure 9.3 it is evident that increasing the cost will affect the results linearly, although at different rates for algorithms with and without candidate selection policies. In this case, a value must be selected that is representative of a real situation, and therefore the fixed transfer cost of 330 milliseconds measured on Sprite [DO91] is as suitable as any.

- **Fixing Threshold:**

  The threshold is fixed at 0 for all experiments, meaning that the algorithms are considering or attempting load distribution at every opportunity. This has the direct implication that the simpler algorithms, such as least loaded, tend to suffer from overdistribution as the workload mobility approaches 1. This effect is illustrated in figure 9.4, where altering the threshold alters the degree of

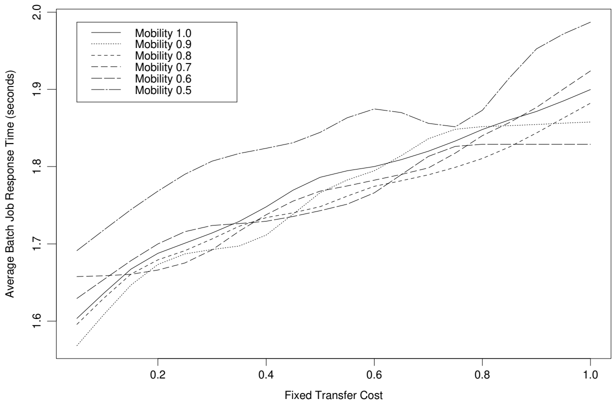


Figure 9.3: *The gradient is constant over a large range of workload mobilities.*

overdistribution.

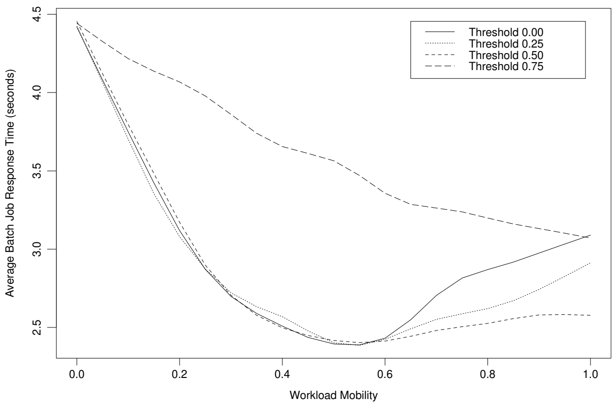


Figure 9.4: *The least loaded algorithm exhibits overdistribution at varying thresholds during the heaviest workload period .*

This reliance on a 'tunable' threshold indicates a less than satisfactory aspect of the more primitive algorithms, where the best threshold varies with the workload. Thus figure 9.4 should not be taken as standard, rather it is simply a nice example of the effect of the threshold on overdistribution.

Given this erratic performance when using the threshold, it seems sensible to use the only the two environmental parameters, which are more consistent over the range of workloads, to evaluate the load distribution algorithms.

9.3.1 Prediction

The prediction mechanism[5] is common to all algorithms except for random, least loaded and hbd-mig. It is used at two different levels, firstly for estimating job and process lifetimes (average and ll-mig), and secondly for estimating the full resource demands of a job or process (wf-ip and wf-mig). There is no training prior to the start of the trace period, and previously unseen processes are always executed locally. A caveat is that this may initially discriminate against the algorithms using the prediction mechanism.

9.3.2 Graphs

All graphs were produced using the Splus statistical package, and consist of twenty data points for each algorithm, smoothed using running medians. Consequently, different neighbouring points (context) may produce a different smoothed value, and this is responsible for the differences between plotted values where the parameters coincide.

9.3.3 The *Process Mix* Hypothesis

To restate, the hypothesis is:

- **Process Mix:** The distribution of a balanced mix of CPU, memory and IO bound processes to each host will lead to the more effective use of resources and delay the onset of bottlenecking.

To support this hypothesis, a more effective use of resources needs to be shown by the algorithms which maintain a balanced *process mix*. This is most visible in a reduction of the average response time compared to algorithms which do not maintain a balanced *process mix*.

The *process mix* of a system can be maintained using either initial placement or process migration, and therefore both of these approaches need to be examined to fully, to determine the support for the hypothesis. For reasons of clarity I will treat them separately, and leave the question of which transfer mechanism performs better, until the discussion of the *initial placement* hypothesis in section 9.3.5.

[5]Defined in section 7.4 page 103

Maintaining the *Process Mix* with Initial Placement

The experiments in appendix A evaluate the performance of the initial placement algorithms.

Results:

- Graphs A.*.1 show that in six of the eight cases wf-ip clearly has the lowest average batch job response time over a mobility range of approximately $0.5 \to 1.0$. Of the remaining two graphs, A.4.1 shows indistinguishable performance between average and wf-ip, while wf-ip has a marginally lower average batch job response time over the mobility range of $0.65 \to 1.0$ in graph A.7.1. Overall, the performance advantage over random is large, and over average the advantage is smaller, but consistently in favour of wf-ip.

  Wf-ip suffers least from overdistribution, showing an increase with a workload mobility greater than 0.8 in only three test sets, as opposed to average, which exhibits an increase in all eight test sets. This suggests that the wf-ip algorithm is less susceptible to the threshold parameter.

- Graphs A.*.2 show that the standard deviation of the batch job response times is reduced as the workload mobility increases, although again, in some situations there is an increase when the algorithm overdistributes. However, wf-ip is only better in three graphs, with the remaining five outcomes undecided.

  One anomaly worth noting, is that graph A.7.1 shows a reduction in the average response time of approximately 50%, however graph A.7.2 only shows a reduction in the standard deviation of approximately 25%. This is substantially less than the expected improvement based on the average response time.

- Graphs A.*.3 show two clear groups of algorithms, those with and those without candidate selection policies. The three algorithms with candidate selection use the same policy, and exhibit a very similar number of remote executions.

- Graphs A.*.4 show that the two classes of algorithms identified in A.*.3 have correspondingly different gradients as the fixed transfer cost increases. The gradients of the algorithms without a candidate selection policy are greater, and perform worse than no load distribution at a lower fixed cost, examples are graphs A.2.4, A.6.4, and A.8.4.

The only other notable feature between the five algorithms is shown in graph A.3.4, where the two predictive fitting algorithms (wf-ip and ff-ip) are more stable than the others. Graph A.3.3 shows that average initiates roughly the same number of remote executions as the two fitting algorithms, and therefore the candidate selection policy is not at fault, but rather the location selection policy. However, one result is insufficient to draw a firm conclusion.

- Graphs A.*.5 in indicate that the standard deviation of the response times increases only minimally as the fixed transfer cost increases. There is no obvious separation of the algorithms based on their candidate selection policies as was evident in graphs A.*.4.

Conclusions:

Random, while gaining a large improvement in performance over no load balancing, produced less consistent and poorer average response times than wf-ip. Average represents the closest competitor to wf-ip, tying one test set and remaining within five to eight percent in all but two of the remaining sets. The best performance for wf-ip was approximately fifteen percent better than average, evident in two test sets at workload mobility of 1.

Thus the performance premium is not high, however, it is sufficient for supporting the hypothesis. In practical terms, the algorithm offers better consistency over a range of workloads, and is less susceptible to overdistribution.

Thus I will conclude that the *process mix* hypothesis is supported for initial placement.

Maintaining the *Process Mix* with Process Migration

The flexibility of process migration encourages a larger number of possible algorithms, and choosing which is more effective requires them to be evaluated on the testbed.

Selecting a wf-mig Algorithm

The numbering of the wf-mig algorithms shown in graphs appendix B does not relate directly to their order in chapter 8, the translation to the experimental numbering is shown in table 9.3.

Name	Version	page	designation
Highest Contributing	1	129	-
and lifetime	2	129	wf-mig 1
and migration cost	3	130	wf-mig 2
and estimated correction	4	130	wf-mig 3
Fairest	-	128	wf-mig 4

Table 9.3: *The experimental numbering.*

Results:

- Graphs B.*.1 show that wf-mig 4 has the worst response time for five of the eight test sets. Algorithm wf-mig 3 has the lowest response times in graphs B.4.1 and B.5.1, with the remaining graphs providing no distinction between the three highest contributing candidate algorithms.
- Graphs B.*.2 offer little distinction between the algorithms.
- Graphs B.*.3 show that wf-mig 4 consistently migrates the fewest number of processes.
- Graphs B.*.4 show that the behaviour of wf-mig 4 is inconsistent, with four graphs having different gradients to those of the highest contributing candidate algorithms. The remaining four graphs show consistent gradients.
- Graphs B.7.5 has a peculiar response for wf-mig 4, otherwise graphs B.*.5 show little helpful in distinguishing the algorithms.

Results:

It is difficult to choose between the highest contributing candidate algorithms, as all generally have very similar levels of performance. However, as wf-mig 3 shows better response time performance in two traces, it is this algorithm that is the better candidate with which to test the hypothesis. I therefore will use wf-mig 3, which is the highest contributing candidate algorithm with the use of estimation to adjust the current behaviour with the historical behaviour. For all later experiments I will simply refer to wf-mig 3 as *wf-mig*.

Investigating the *Process Mix* Hypothesis

Appendix C contains the results for the comparison of wf-mig against the two non *process mix* maintaining algorithms.

Results:

- Graphs C.*.1 show wf-mig produced consistently lower average batch job response times than both competitors in seven of the eight test sets, over the $0.2 \rightarrow 1.0$ range of workload mobilities. The only marginal result was for test set eight, where wf-mig was better over the more restricted mobility range of $0.2 \rightarrow 0.8$.
- Graphs C.*.2 produce no conclusive results for any one algorithm.
- Graphs C.*.3 show that wf-mig consistently has the highest number of migrations.
- Graphs C.*.4 show that all three migration algorithms have similar gradients in response to the increase in the fixed transfer cost, even with wf-mig migrating more processes.
- Graphs C.*.5 produce no conclusive results for any one algorithm.

Conclusions:

The wf-mig algorithm shows a consistently better average response time in graphs C.*.1, over a large range of workload mobilities. The higher number of migrations performed by wf-mig does not reduce its performance with respect to the other migration algorithms, as the fixed transfer cost is increased. Therefore these results strongly support the process mix hypothesis for process migration.

All algorithms reduce the standard deviation of response times, and no one algorithm stands out as performing better in this regard.

9.3.4 The *Prediction Accuracy* Hypothesis

- **Prediction Accuracy:** Perfect prediction accuracy is not required nor realistic, a reasonable estimate is sufficient for good performance.

Average is quite a weak predictor when applied to workload resource demands, as it exhibits a large standard deviation [LO86, DI89]. However, the results of test sets A.* show that even this weak form of prediction gives good performance when coupled with wf-ip. Whilst this supports the hypothesis by implication, the relationship between prediction quality and performance is still unknown. This deserves further work and is discussed further in chapter 10.

9.3.5 The *Initial Placement* Hypothesis

- **Initial Placement:**

  Initial placement can use prediction to attain performance comparable to that of process migration.

The previous sections have found that both process migration and initial placement are improved by maintaining a balanced *process mix*. Graphs D.* compare these two approaches, and also show the results of a combined metric which is discussed later in section 9.3.6.

Results:

- Graphs D.*.1 show wf-ip has the lowest response time with four test sets, wf-mig produces the lowest response times in two, while the remaining two are too marginal to decide.
- Graphs D.*.2 show that wf-ip consistently gives the lowest standard deviations.
- Graphs D.*.3 show widely different behaviour between the number of candidates selected by wf-ip and wf-mig. This is inline with the expected behaviour due to the different types of distribution candidates selected by the migration and initial placement algorithms. These graphs show that the number of remote executions generated by wf-ip decrease much more rapidly than the number of migrations initiated by wf-mig, and in three cases, wf-ip eventually transfers fewer jobs than wf-mig migrates. The interesting point is that even though the initial placement algorithm transfers fewer candidates, the performance is no worse than process migration.

- Graphs D.*.4 show no substantial difference in gradient between wf-ip and wf-mig.
- Graphs D.*.5 show that in six cases, wf-ip has dramatically better standard deviation results, up to 25% as the fixed transfer cost increases.

These results demonstrate that wf-ip does indeed perform at least as well as wf-mig, in both the average and standard deviation of response times. This conclusion also extends to the other initial placement algorithms which perform almost as well as wf-ip, showing that they are often better than the non *process mix* maintaining process migration algorithms.

9.3.6 The *Combination* Hypothesis

- **Combination:** Combining initial placement and process migration in a single load distribution policy, will enable placement errors to be corrected, thus improving system performance.

The combination hypothesis does not have a specific algorithm, instead the two best performing initial placement and process migration algorithms (worst fit) were run side by side.

Graphs D.1.* though D.8.* show the reaction of the combined algorithm, along with the wf-ip and wf-mig components.

Results:

The first observation is that the combined algorithm is often the average between the wf-ip and wf-mig algorithms. The only real exception is with regard to the average response time for D.7.1, where the combined algorithm results in a lower average response time than either of its component parts. Although this test set is clearly an exception, there is also a little promise evident in the graphs D.4.1 and D.6.1. This may indicate that there is perhaps some scope for future work on this - with a less naive combination of the two algorithms, however the advantage seems minimal.

The number of transfers carried out by the combined algorithm is substantially lower than the number required for wf-ip, and only slightly more than the number of transfers carried out by wf-mig. However, at some points the number of transfers is greater than either of the two component algorithms, perhaps an artifact of the simple integration of the algorithms .

The only conclusion possible is that there is insufficient evidence either way, and future work is required to resolve this problem.

9.3.7 Additional Observations

There are a number of observations that can be made about the algorithms and results which are not directly related to the support of the hypotheses.

- Candidate selection is important and increases in significance as placement costs increase.
- The average response time increases linearly with respect to the transfer cost over the range of 50 to 1000ms.
- The results confirm Zhou's [Zho87] finding that the average response time is concave and decreasing, as mobility increases, although with the additional note that overdistribution can result in an increase as the workload mobility approaches 100%.

9.4 Conclusions

The following conclusions summarise the findings of this chapter. They relate directly the the hypotheses from chapter 3 and the results are only valid for the distribution of heavyweight processes between a small number of hosts. Chapter 10 applies these specific results to more general issues in load distribution.

- **The *process mix* hypothesis is supported for both initial placement and process migration:**

  The reduction in average response times show that the *process mix* hypothesis is supported for initial placement, and strongly supported for process migration over a wide variety of workloads.

- **The *prediction accuracy* hypothesis is supported:**

  Good performance was gained using a weak predictor, supporting the hypothesis.

- **The *initial placement* hypothesis is supported:**

  Initial placement is as good, if not better than migration if the sole aim is to improve the performance of a distributed system. The extra effort in implementing a full process migration system is not reflected in the resulting performance gain over initial placement for the process based system model studied.

  Thus process migration of heavyweight processes appears to be more suitable for other requirements, such as migrating servers or user shells for reliability and avoidance of hot points, or migrating to enforce ownership rights, and other such policy oriented decisions.

- **The *combination* hypothesis may apply during heavy workloads:** However, in general it offers no real advantage over initial placement or process migration.

Chapter 10

Conclusions and Future Work

The first section of this chapter describes issues still requiring further investigation. The second section discusses the major results of this work and general issues of load distribution.

10.1 Future Work

The results presented in chapter 9 showed that maintaining the *process mix* in a homogeneous distributed system offers advantages consistency, and therefore predictability of service times, over simpler systems such as random and least loaded. In comparison to more complex load distribution policies such as those using averages, maintaining the process mix constitutes only a small increase in state and complexity, and is sufficiently promising to warrant further study.

10.1.1 Other Algorithms

The wf-ip algorithm is not the only way to maintain the *process mix* of a distributed system, and other algorithms may offer additional benefits.

The means of limiting the amount of load distribution is often a simple load threshold mechanism, however this is at odds with the maintenance of a balanced *process mix*, whereas a balance criteria would be a more suitable means of controlling the degree of load distribution.

Future work:

- Other algorithms for maintaining the *process mix* of a distributed system should be investigated.
- The use of a balance criteria to reduce the amount of distribution taking place for worst fit algorithms should also be investigated.

10.1.2 Different Numbers of Hosts

The experimental work in this thesis has only considered a Xterminal-Server model with five servers. These results need to be investigated with respect to networks of different sizes and configurations.

Future Work

- Networks of different sizes may respond differently to the maintenance of the *process mix*. This aspect should be explored, although I suspect that the technique will be most successful with a small to medium number of large servers.

10.1.3 Update Frequency Dependence and Load Estimation

A side effect of maintaining the *process mix* of a system with initial placement is that the load, after scheduling a job to a host, is estimated during the selection of a destination host. This estimate is also used to extrapolate the load on each host between the periodic load updates, leading to the following hypothesis:

- **Information Exchange:** Good predictions can be used to estimate the host state after a placement, thus a policy using such estimates is less sensitive to the frequency of host load updates.

The current implementation is naive in that there is no consideration applied to the possible completion of jobs within the update period. Instead the system assumes that the update period is sufficiently short to provide a corrected update measured from the real machine before the estimate becomes too inaccurate.

The *Information Exchange* hypothesis suggested that the availability of predicted resource information would allow longer times between load updates before performance is degraded, but this would require a better means of estimating the inter update load, and was not investigated in detail. The following future work would be a prelude to investigating this hypothesis.

Future Work:

- A study to determine the sensitivity of the wf-ip algorithm to poor load estimates would help determine the need for a better system of inter update load estimation.

- If the algorithm is indeed sensitive, a starting point for improving load estimation would be to consider the jobs that finish within the interval. There are many possible methods for achieving this, including the use of decay factors, or by simply subtracting a jobs estimated load contribution from the hosts, after either being notified of its completion or after its predicted lifetime has expired.

10.1.4 Combined Policies

The results of investigating the *combination* hypothesis hinted that at higher system loads, a policy that uses both initial placement and process migration to maintain the *process mix* of a distributed system could perform better than policies restricted to one transfer mechanism. Whilst the evidence in support of this is minimal, there is enough to suggest that this area could warrant further investigation, especially with a less naive algorithm.

Future work:

- Develop a better combined policy. As with estimation, there are many possibilities some of which are refined versions of *the repair approach.* This approach involves the assumption that, as initial placement cannot move processes, a serious imbalance (or distribution error) is best repaired by moving the problem process or processes with process migration. This would require a unified policy, and identification of a problem situations requiring migration.

10.1.5 Prediction Sensitivity

The relationship between the accuracy of job resource predictions and the performance of the wf-ip algorithm is unknown. While this alone is worth investigating, further impetus is added by the observation that the predicted lifetime is crucial in the computation of the estimated utilisations from the estimated resource consumption totals. Thus any inaccuracy in this prediction compounds the inaccuracy of the other predicted values.

Future work:

- Investigate the sensitivity of wf-ip to the accuracy of resource predictions, and if it is sensitive, the investigation should extend to how more accurate predictions can be provided.

10.1.6 Axis Scaling (Resource Priorities)

Resource fitting allows certain resources to be scaled or in other words prioritised. If for example, the system is mismatched between the resources it provides and the resources the workload demands, certain resources may be the cause of bottlenecking earlier than in a system where all resources are in tune with the workload. In this situation, the most critical resources could be scaled to reflect their importance to the performance of the system.

Future work:

- Investigate whether some resources are more significant than others independent of workload, and whether this information can be used to 'tune' wf-ip by scaling the axis during fitting.

10.1.7 Application in Heterogeneous Systems

Work in the area of resource balancing over heterogeneous systems has in the past concentrated on the *capability* of a host to execute a job rather than on the *balance* of the systems resources. There is no reason why *process mix* cannot be applied to distributed systems that are heterogeneous. In fact, there is possibly greater benefit to be gained as now not only the workloads are biased, but the host resources are also biased.

Future work:

- Maintaining the *process mix* in heterogeneous systems may offer even greater benefits. Problems involve computing predictions and loads in machine independent form (see [Bon93]).

10.1.8 Implementation Design

This thesis is intended as a proof of concept, and therefore most attention is focused on the design of policies rather than on their implementation. As with any simulation study, whilst the results look promising, the final proof lies in a real system.

Future work:

- Implement a load distribution system on a real system to verify the usefulness of *process mix* based algorithms such as wf-ip.

10.2 Conclusions

The work in this thesis involves the investigation of load distribution policies, with the intention of extending the role of resources in distribution decisions. The conclusions in section 9.4 relate directly to the specific goals of the thesis. This section uses those results to address two fundamental issues in the field of load distribution.

10.2.1 Process Migration and Initial Placement

Initial placement is shown to be as effective, if not superior to process migration for improving the performance of the distributed system and workload studied. This confirms the finding of Eager et al. [ELZ88] that "It is not worth the effort of migrating *active* processes if the sole intention is to increase performance...", in contrast to the analysis of Downey and Harchol-Balter [DHB95], who question the validity of the modelling and its applicability to modern systems.

In particular, Downey and Harchol-Balter criticise Eager et al. on the following points:

- **System Model:**

  Downey and Harchol-Balter claim that as jobs are initially placed at no cost, Eager et al. implicitly model a "server farm", where jobs have no natural host affinity. As this result is often applied to workstation environments, where processes do exhibit an affinity for hosts, Downey and Harchol-Balter claim that an initial placement cost should be included, otherwise process migration is unfairly handicapped.

- **Workload Model**

  Downey and Harchol-Balter also find fault with the two main aspects of the workload model, namely the batch arrival of jobs, and the distribution of job lengths. The batch arrival of jobs is considered to not apply to modern systems and more importantly, they claim the distribution of process lifetimes is unrealistic, and handicaps process migration.

- **Local Scheduling**

  Lastly, Downey and Harchol-Balter claim that as Eager et al. do not simulate or consider local scheduling, the accuracy of the model is questionable.

They conclude that "the general result of ELZ does not apply to current systems", and that the benefits of process migration should be reexamined.

Whilst these arguments raise reasonable points, none apply to this study, which confirms the findings of Eager et al. [ELZ88] are correct for at least the Xterminal server model.

10.2.2 The Complexity of Initial Placement Policies

Eager et al. [ELZ86b] concluded that "...extremely simple adaptive[1] load sharing policies, which collect very small amounts of system state and which use this information in very simple ways, yield dramatic performance improvements. These policies in fact yield performance close to that extracted from more complex polices whose viability is questionable".

Random does indeed perform well, confirming the first part of their conclusion. However the more complex policies (least loaded, average, ff-ip and wf-ip) performed sufficiently better to warrant further consideration, particularly in light of random's poor performance as the cost of initial placement increases[2].

This result is anticipated in part by Eager et al. who suggest that any practical implementation would only place jobs with a transfer cost less than 10% of the processing cost, that is, they would introduce a candidate selection policy. This not only means both more state and complexity, but in practice random performed very poorly when coupled with a candidate selection policy, see section 8.1.3.

In conclusion, it appears that the more complex initial placement policies have been overlooked, and offer real performance advantages over simpler policies.

10.2.3 Consideration of Resources

The advantages of considering the available system resources, and matching those to the resources required by jobs are clear. In particular, whilst the improvement of wf-ip over

[1] Termed *Dynamic* in this thesis.

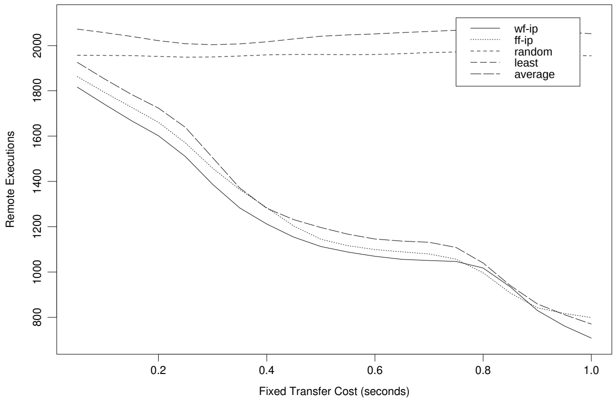
[2] Recall that probing did not improve the performance, due perhaps to the small number of hosts.

average is not large, it is nonetheless consistent, and significantly, the additional state and complexity of wf-ip is minimal.

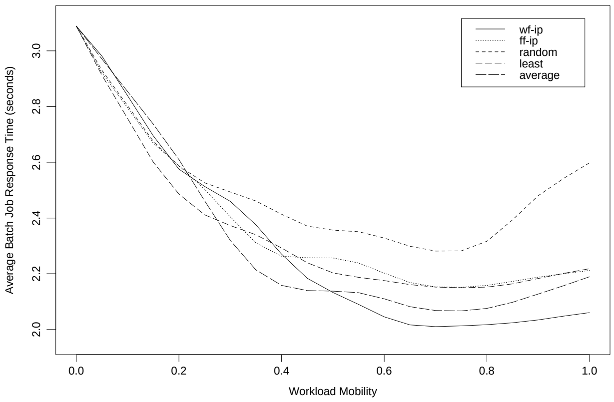
Thus, resource based load distribution polices are worthwhile for the type of system investigated, and offer improved and consistent performance with little additional complexity.

Appendix A

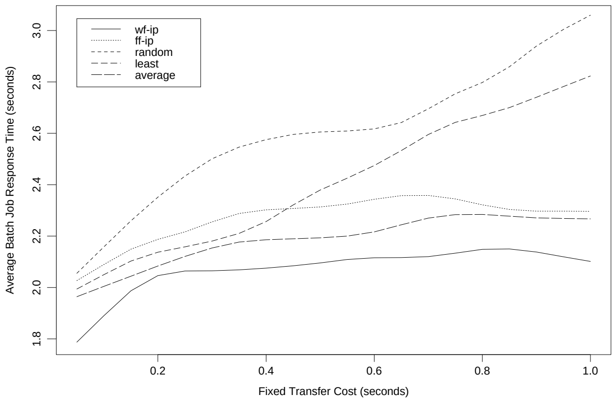
Initial Placement



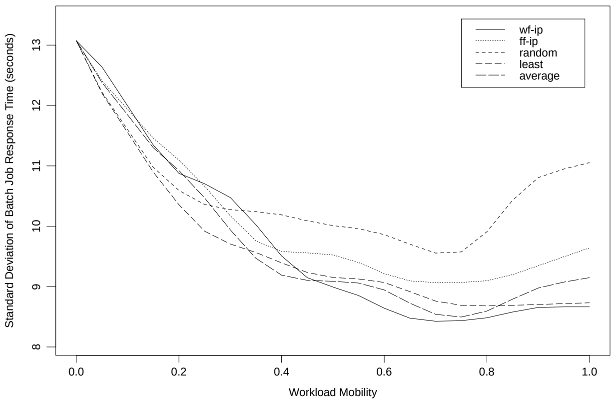
3.



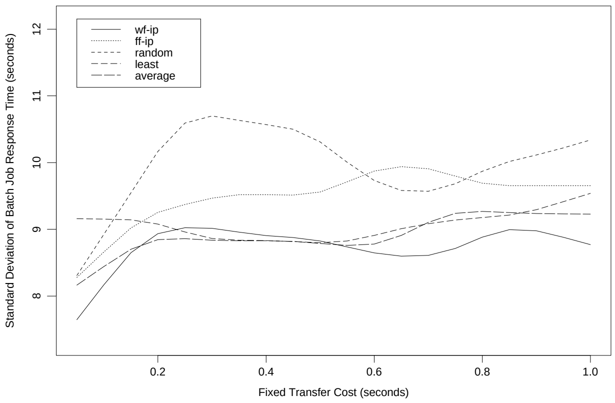
1



4

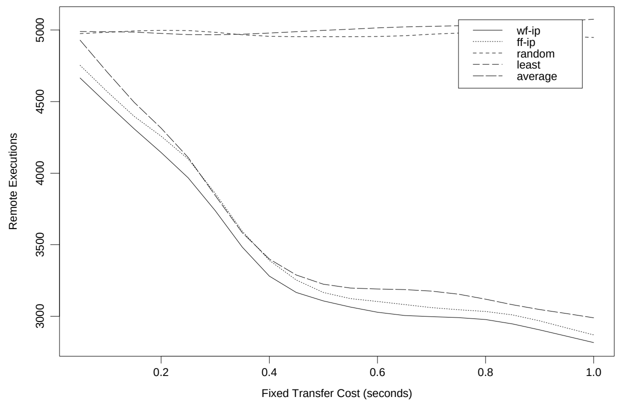


2

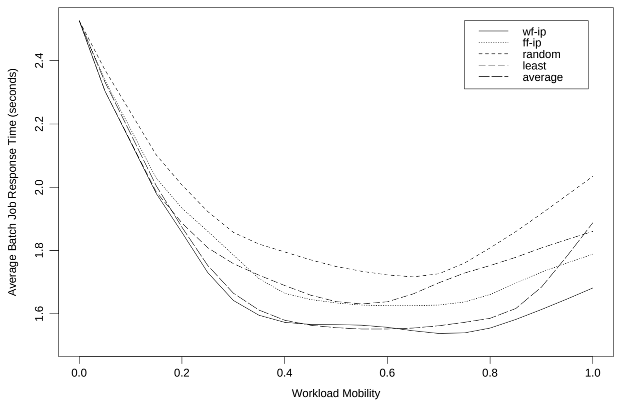


5

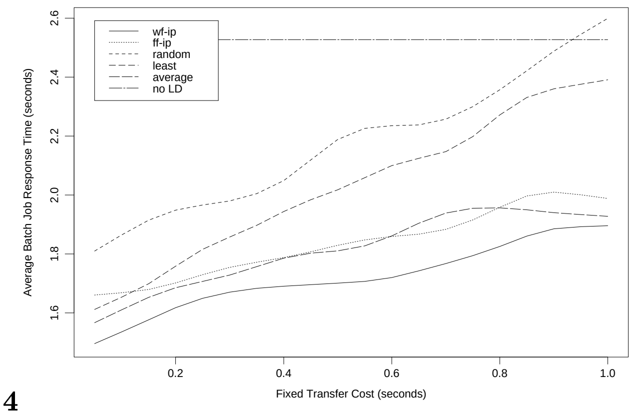
Figure A.1: *CPU bias.*



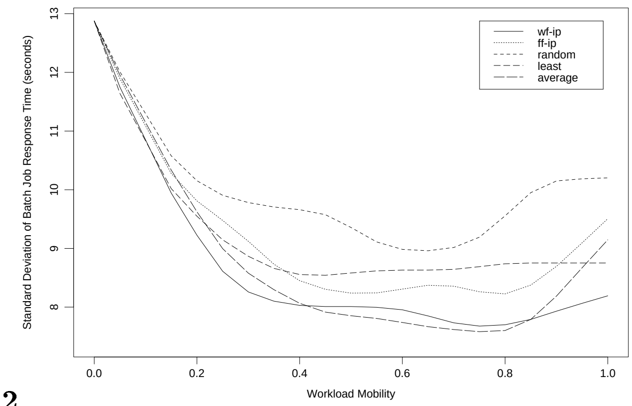
3.



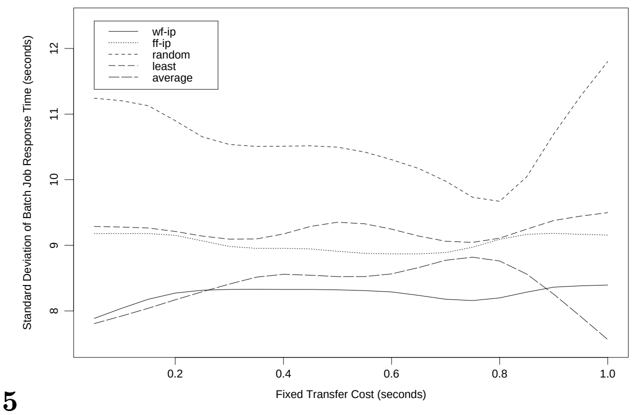
1



4



2



5

Figure A.2: *Memory bias.*

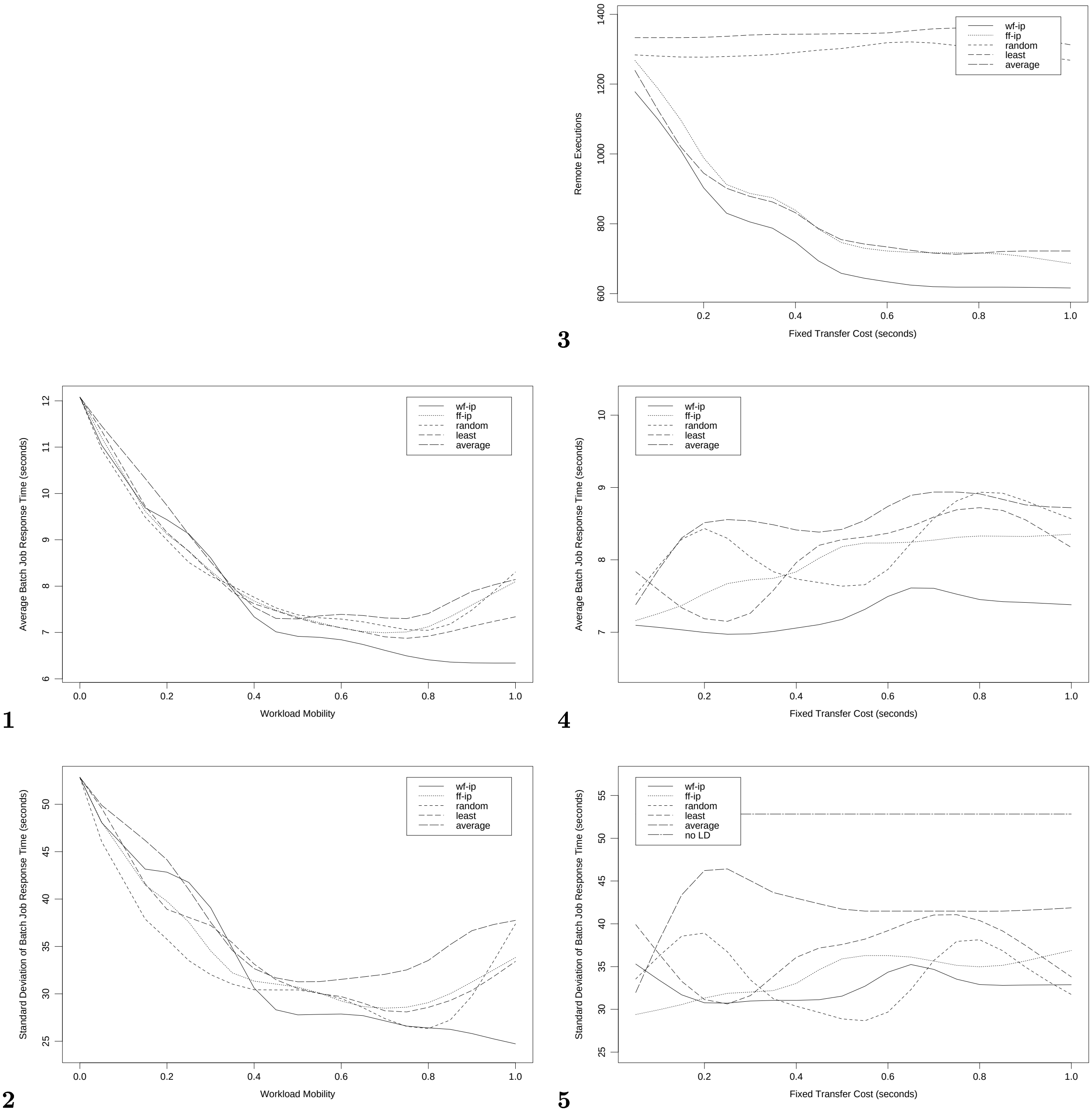


Figure A.3: *IO bias.*

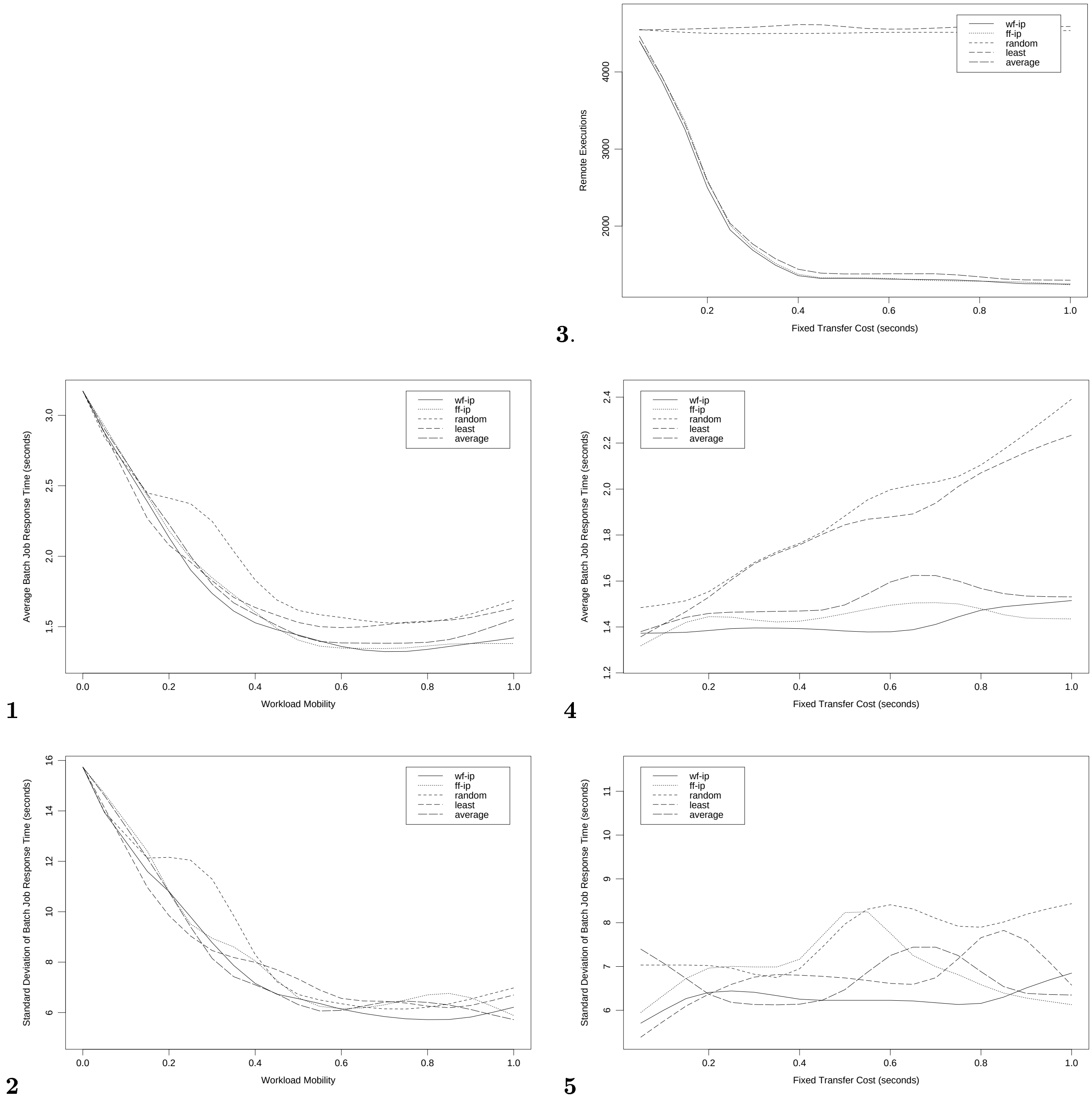
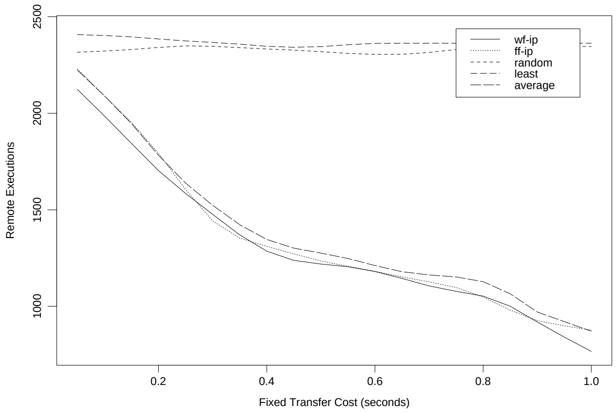
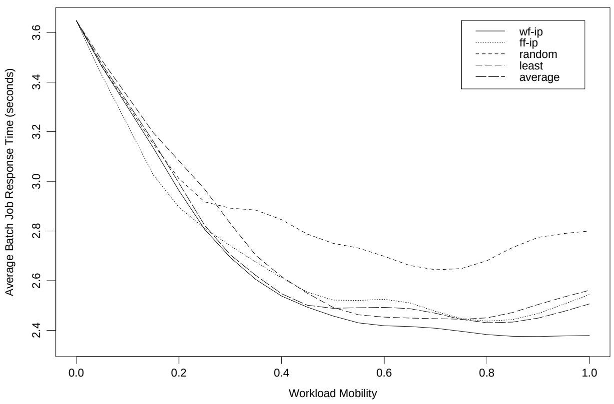


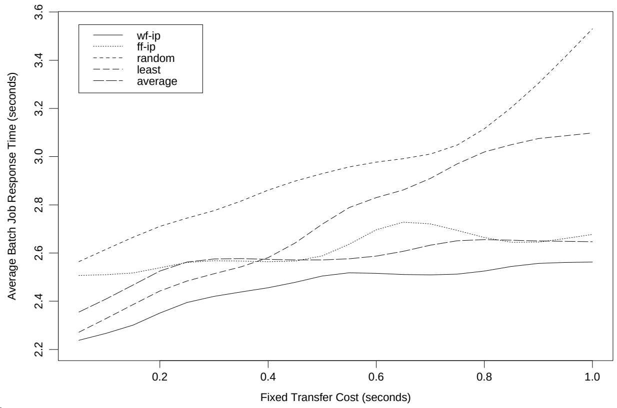
Figure A.4: *CPU and memory bias.*



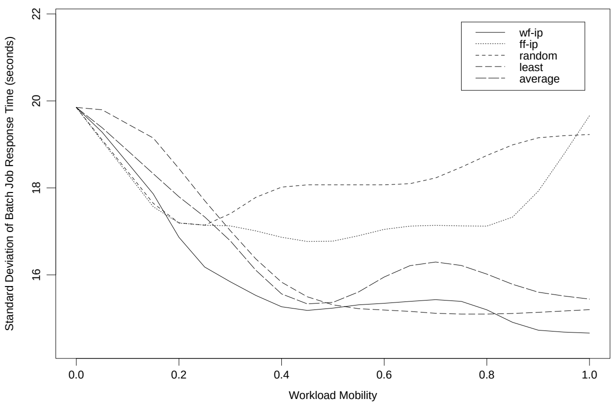
3.



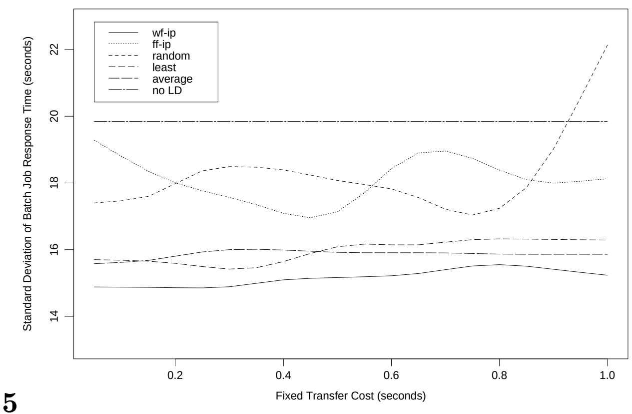
1



4

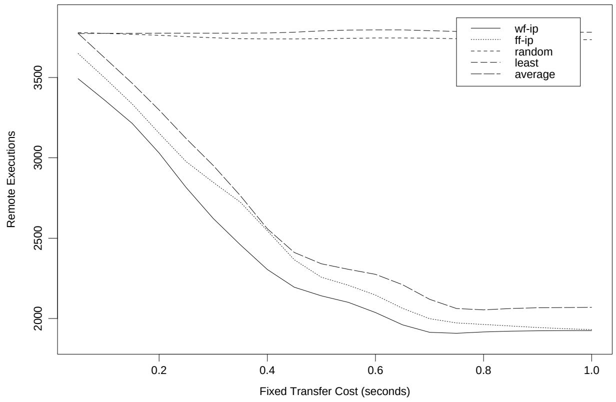


2

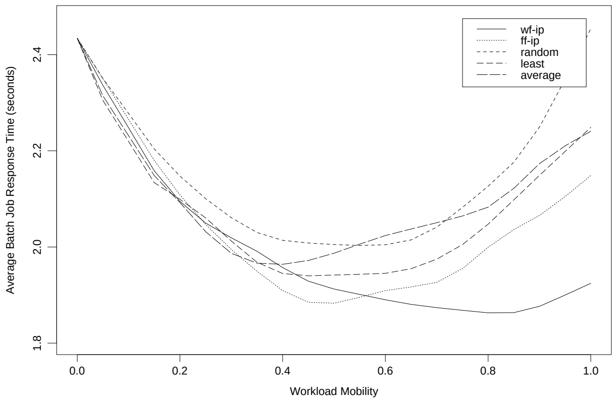


5

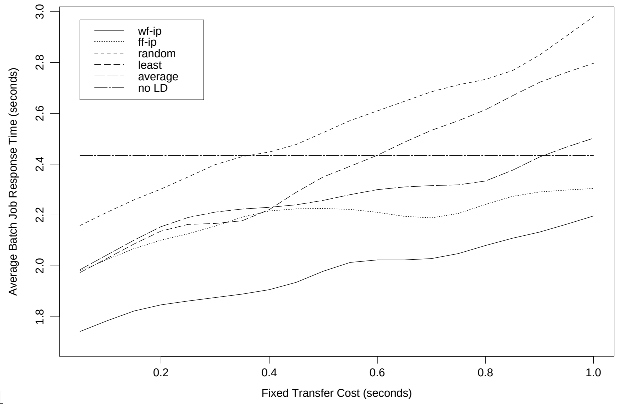
Figure A.5: *CPU and IO bias.*



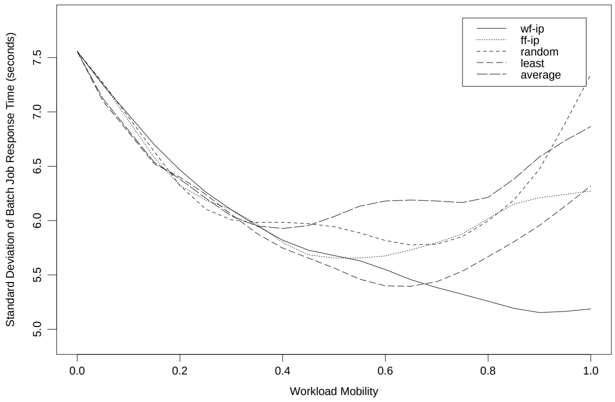
3.



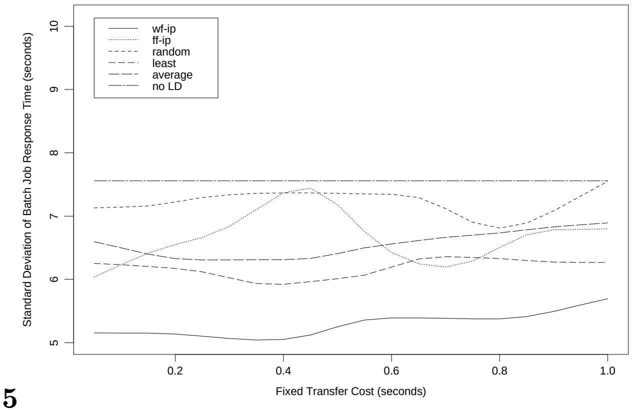
1



4

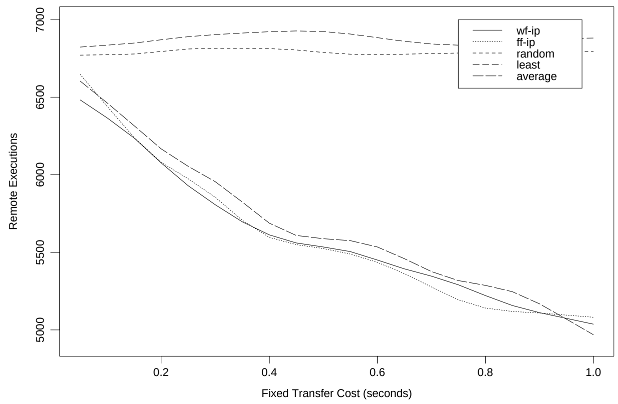


2

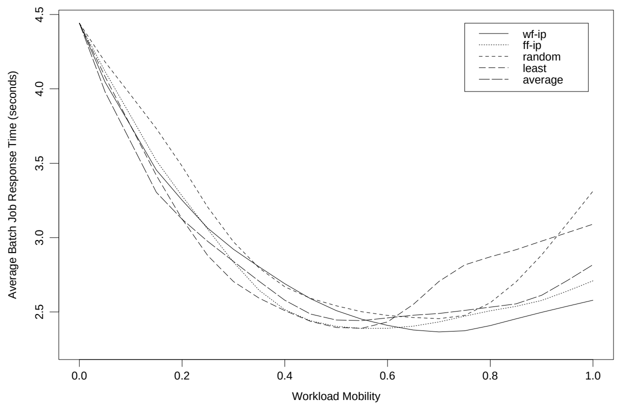


5

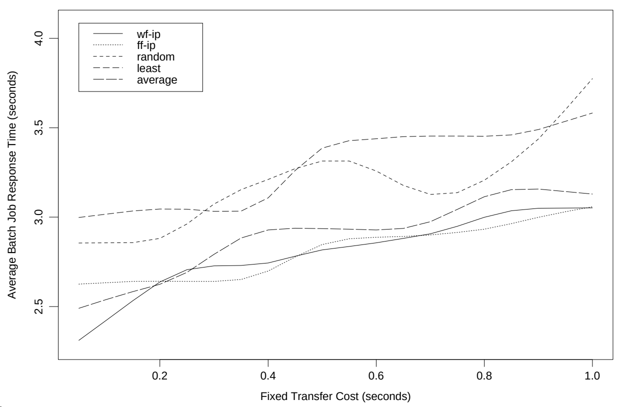
Figure A.6: *Memory and IO bias.*



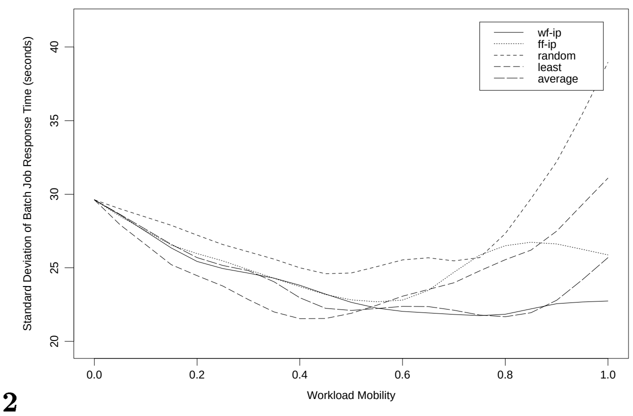
3.



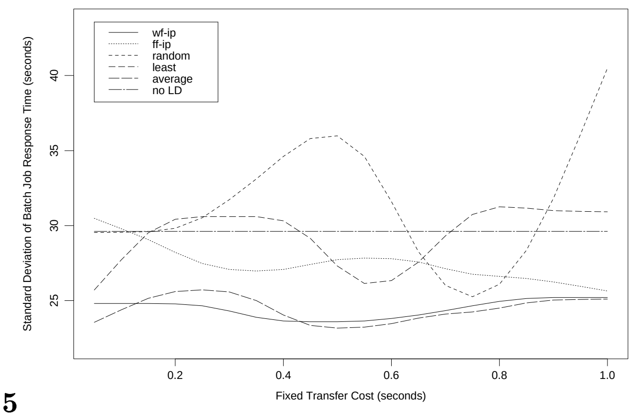
1



4

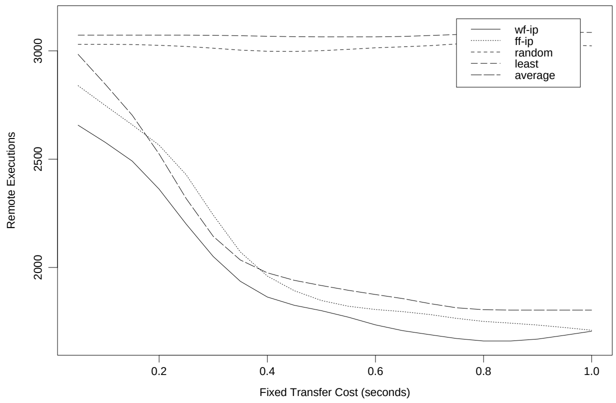


2

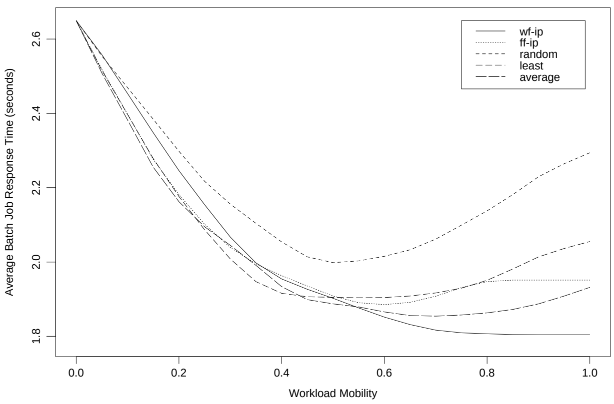


5

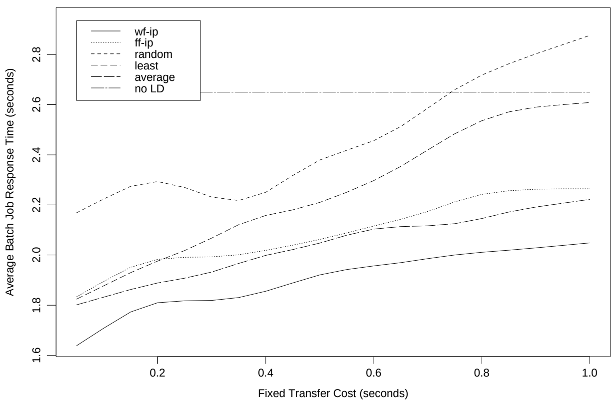
Figure A.7: *Highest load, no bias.*



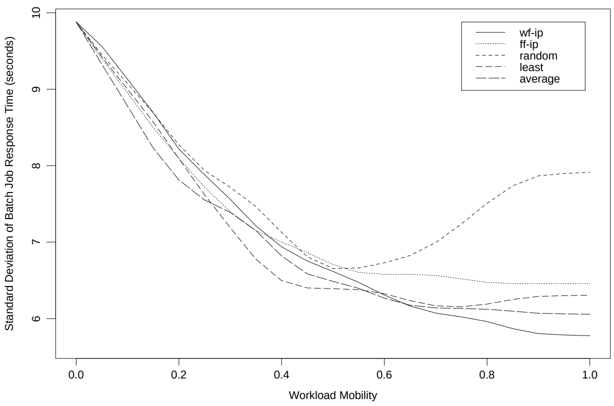
3.



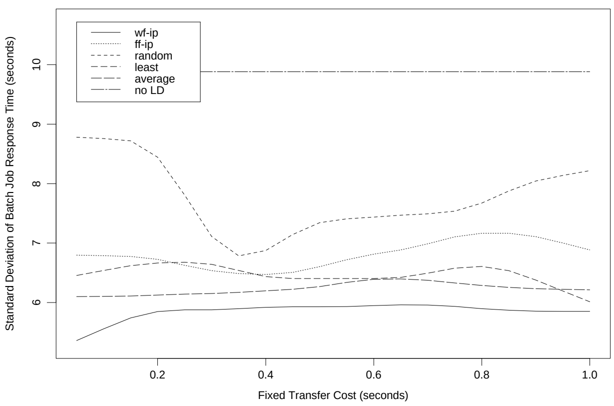
1



4



2

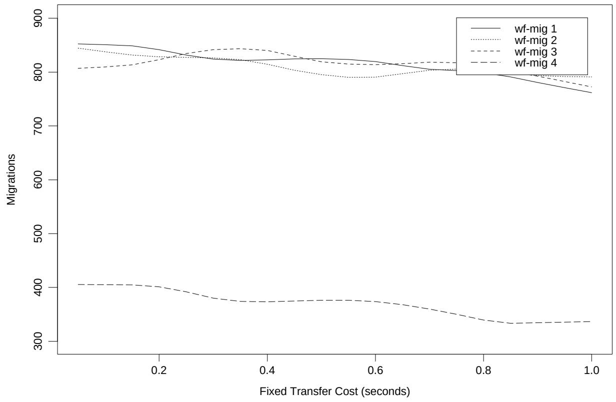


5

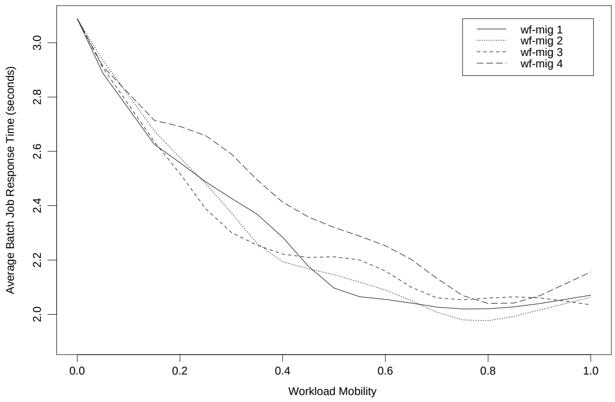
Figure A.8: *Medium load, no bias.*

Appendix B

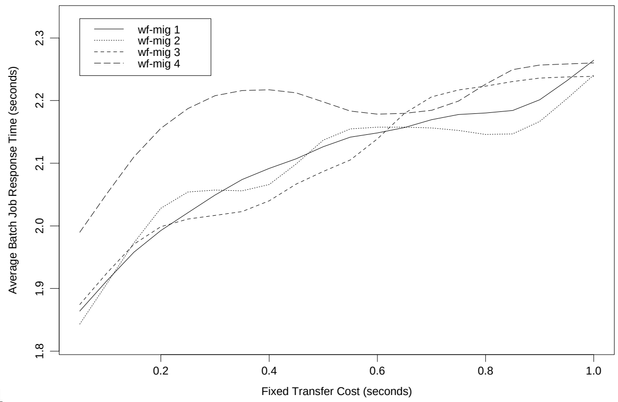
WF Migration



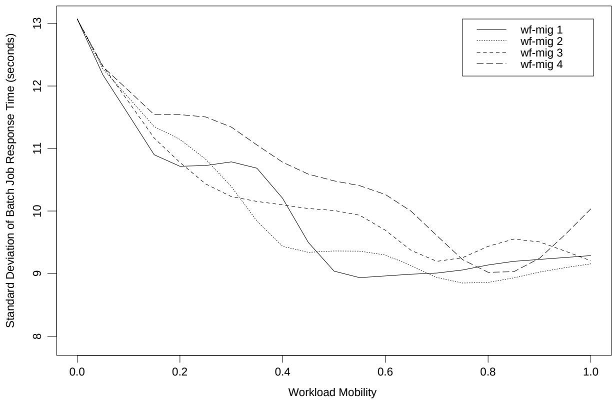
3.



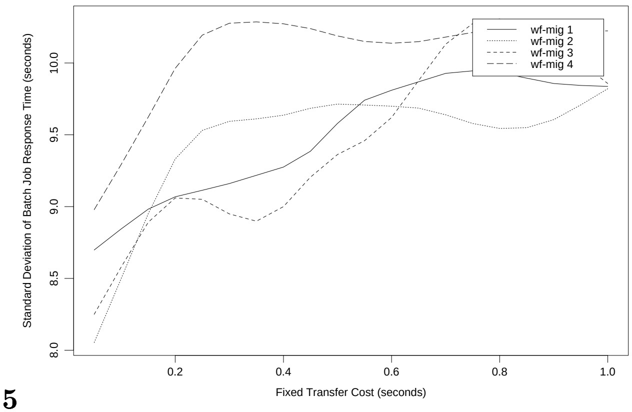
1



4



2



5

Figure B.1: *CPU bias.*

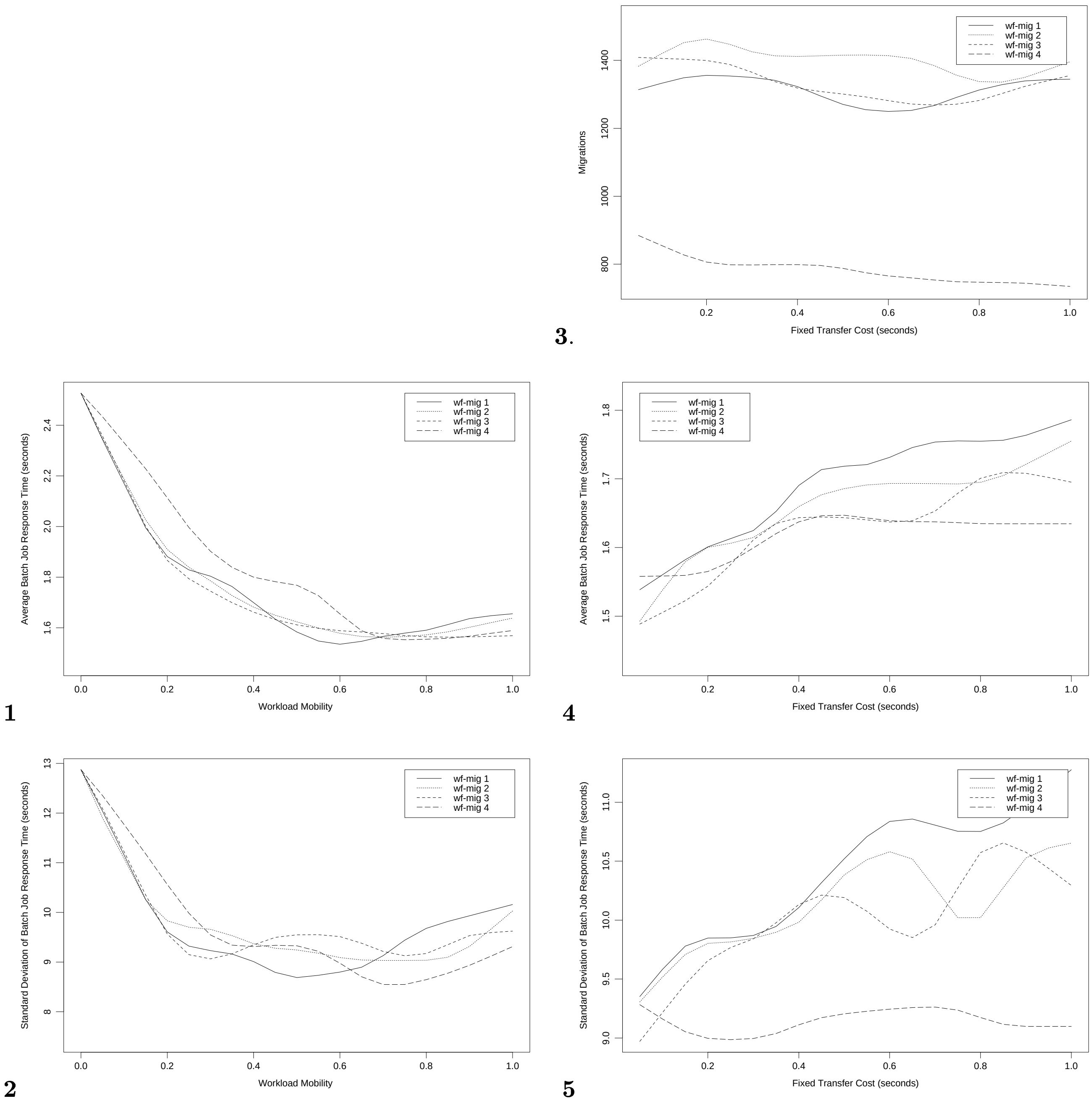


Figure B.2: *Memory bias.*

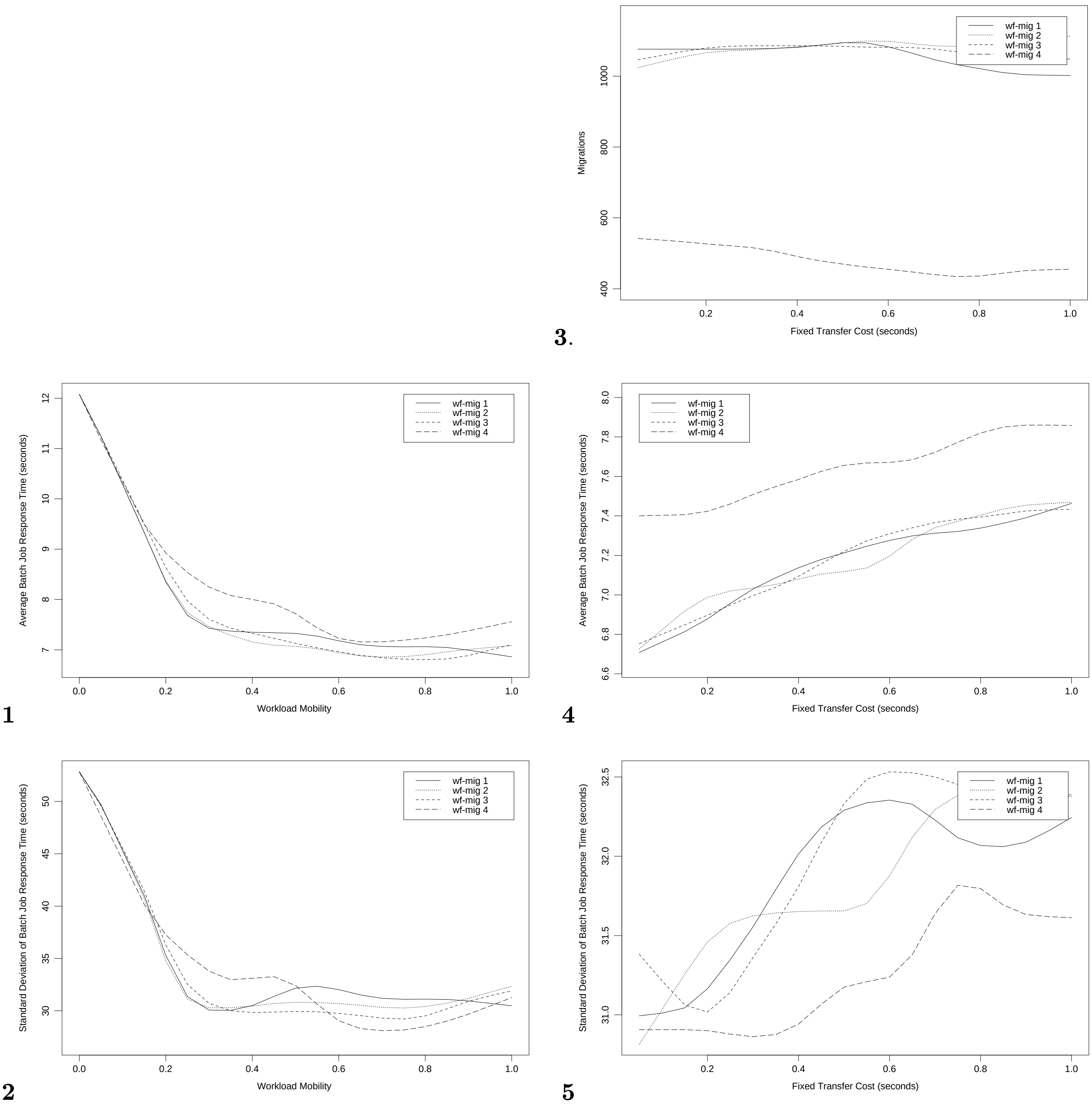


Figure B.3: *IO bias.*

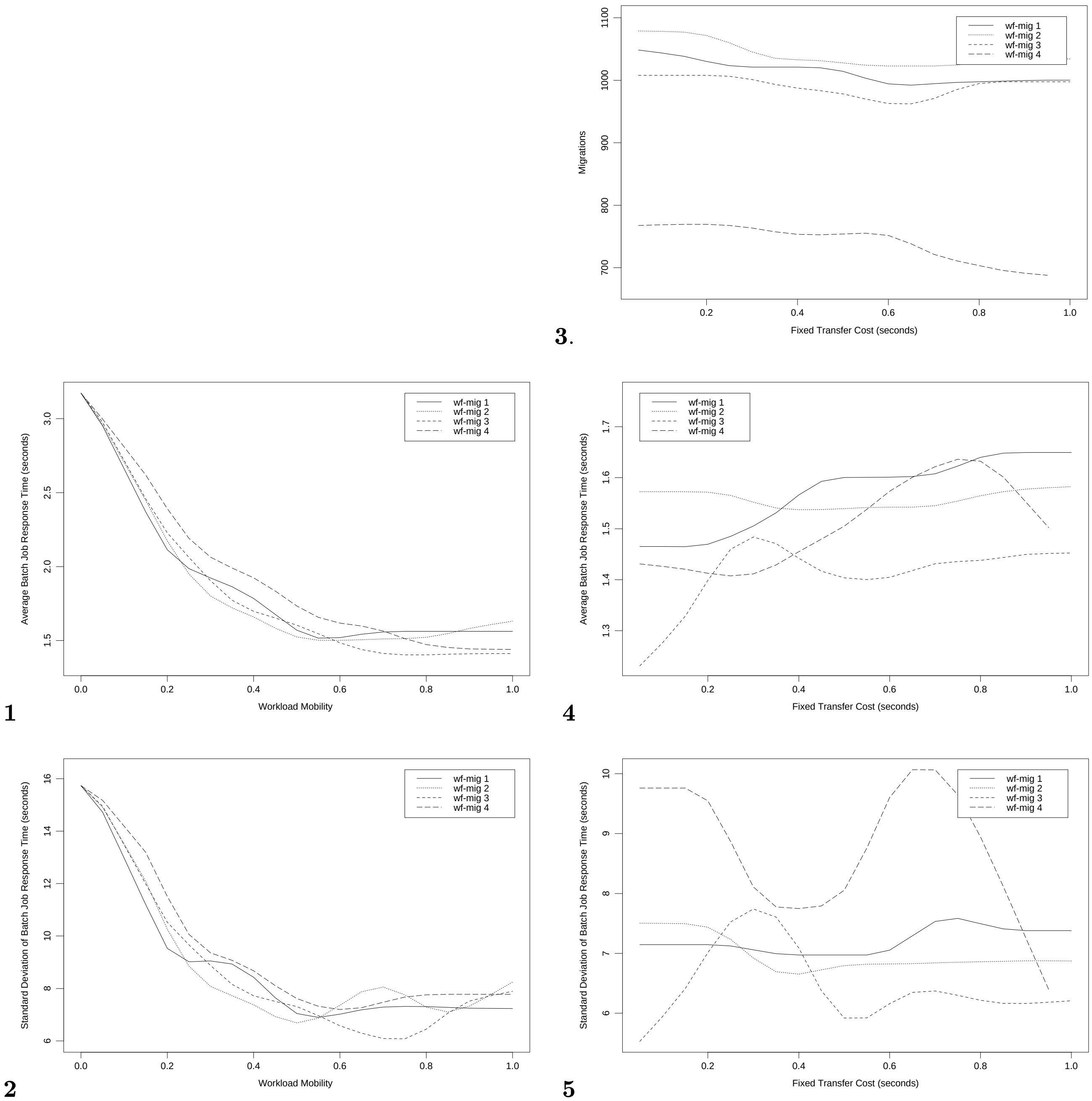


Figure B.4: *CPU and memory bias.*

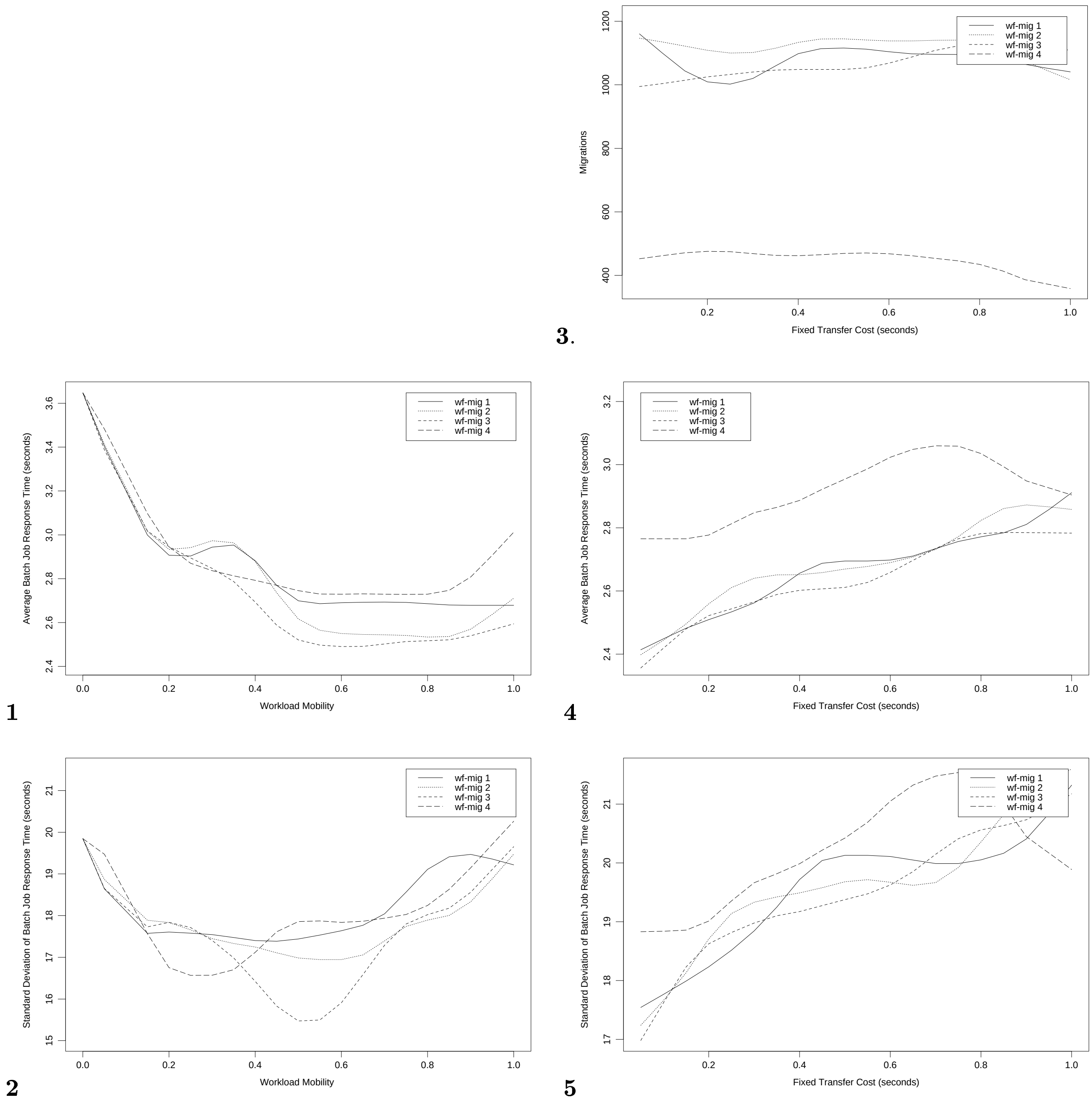


Figure B.5: *CPU and IO bias.*

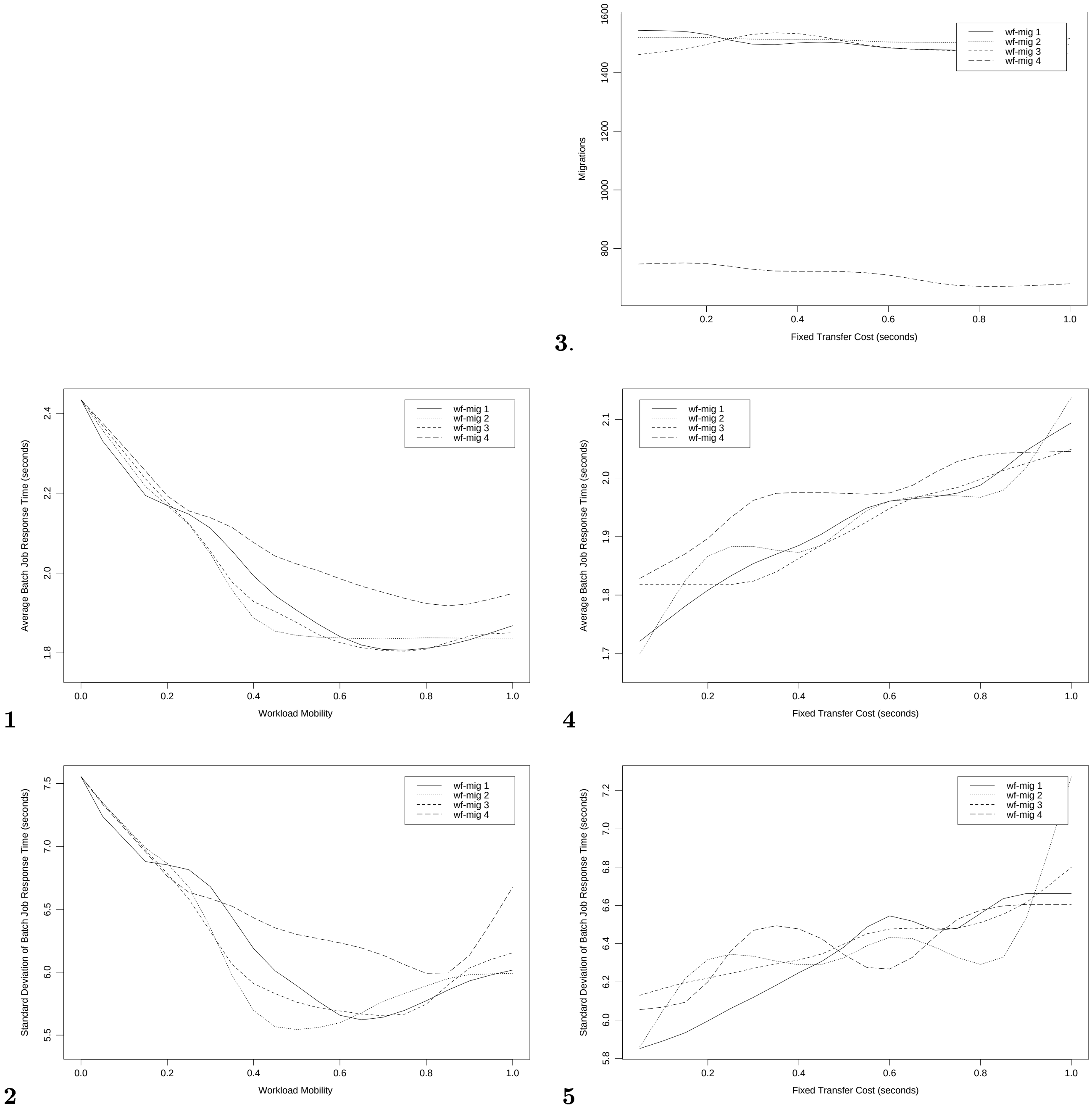


Figure B.6: *Memory and IO bias.*

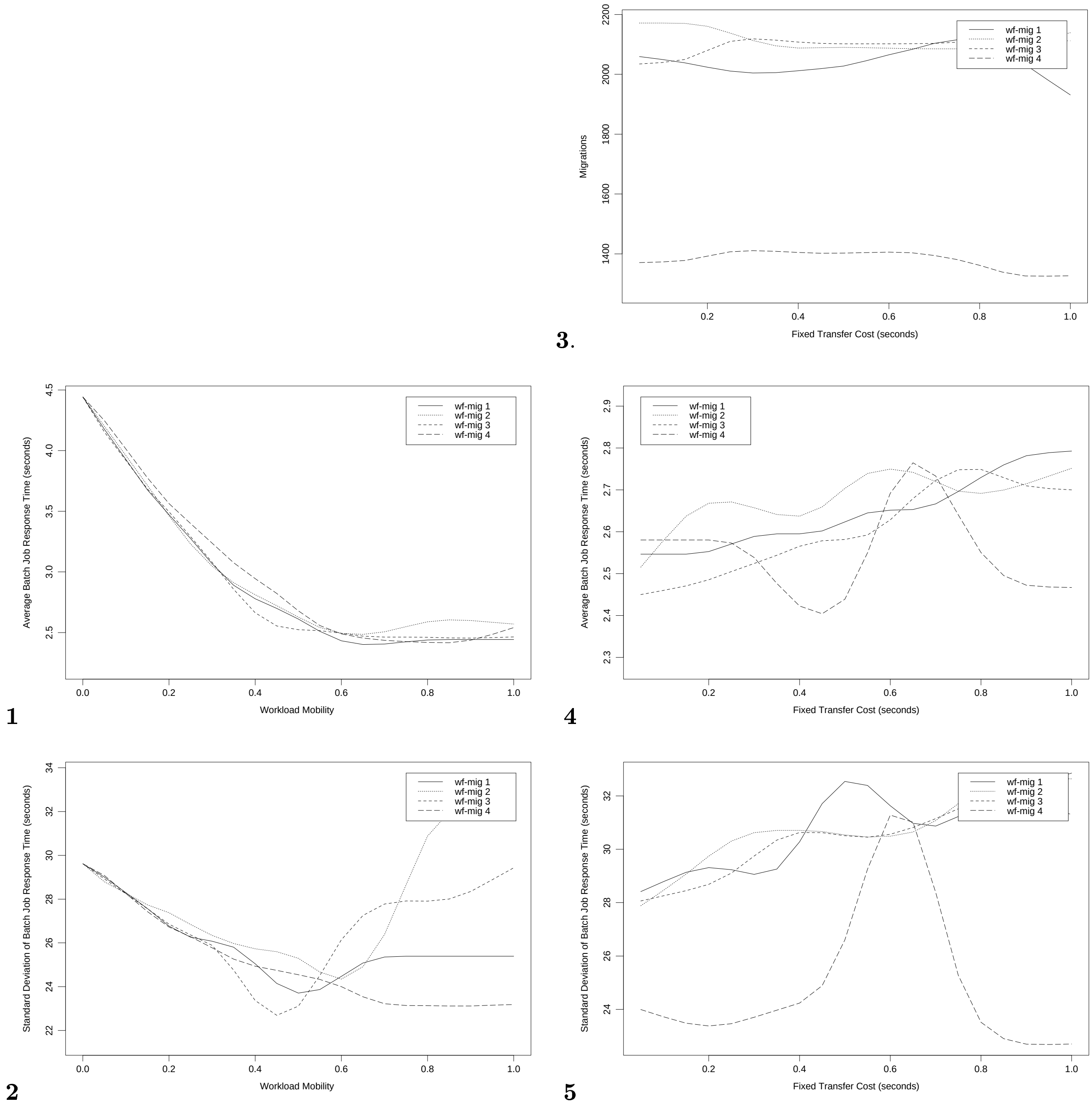


Figure B.7: *Highest load, no bias.*

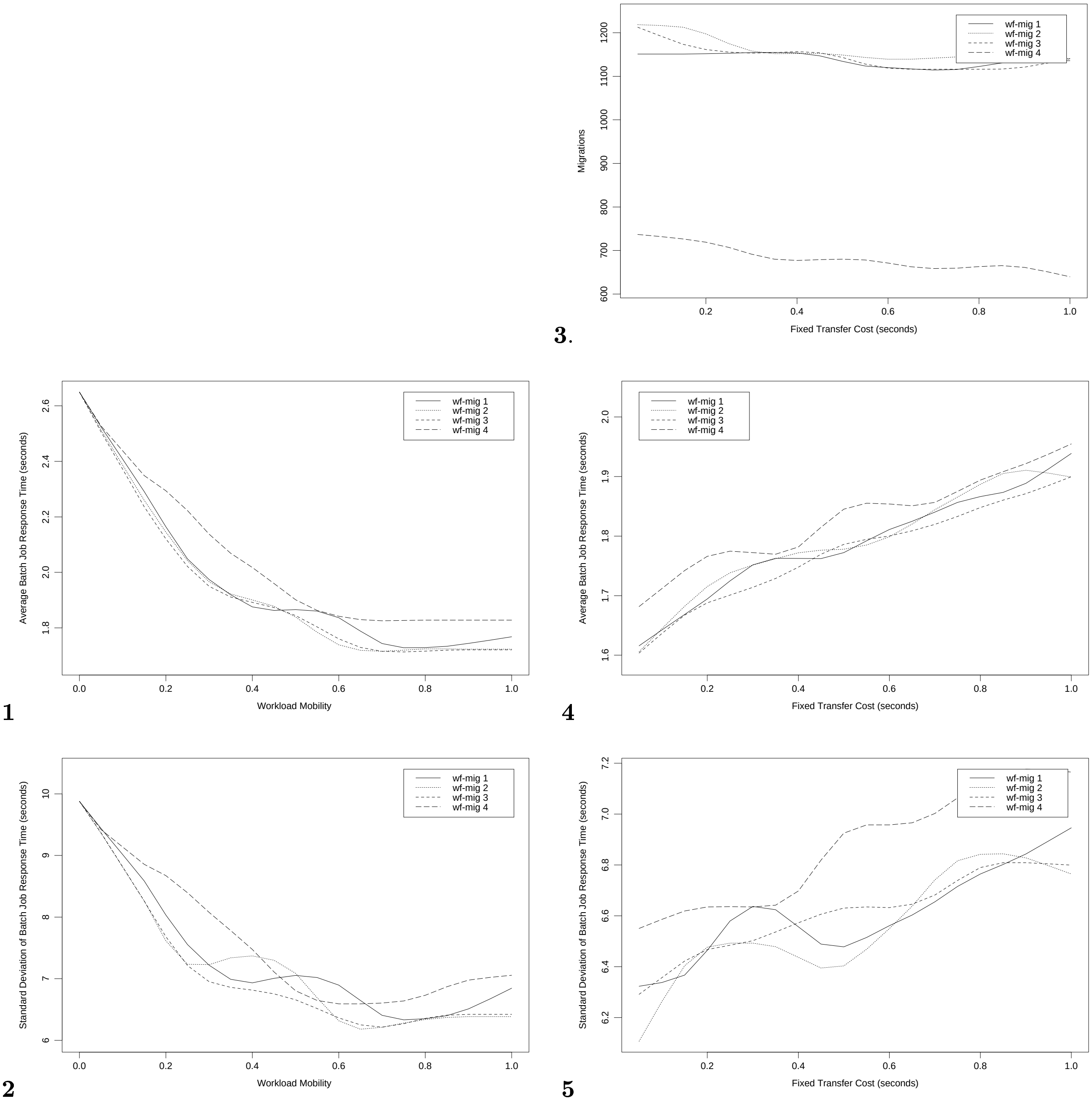


Figure B.8: *Medium load, no bias.*

Appendix C

Process Migration Results

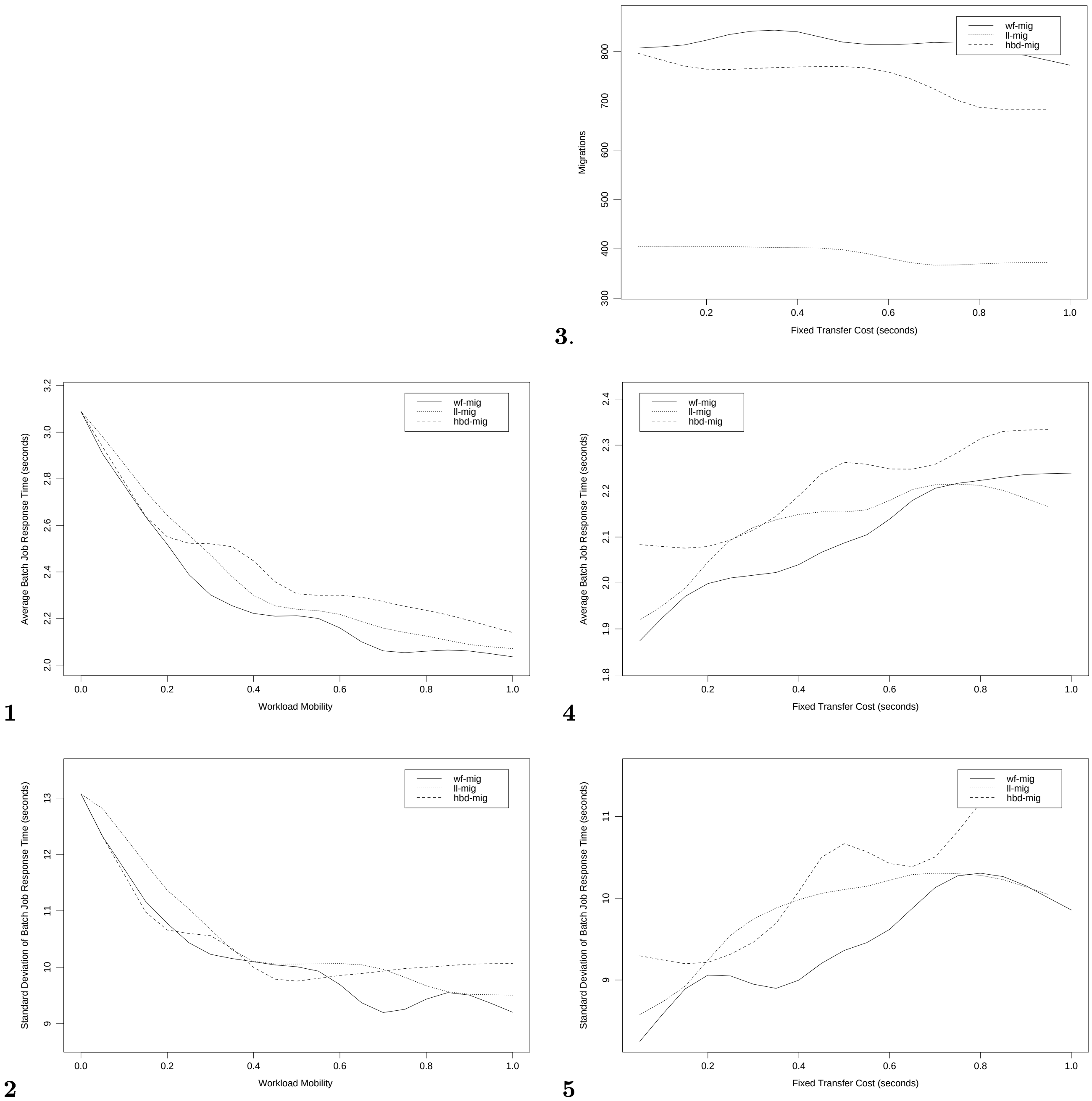


Figure C.1: *CPU bias.*

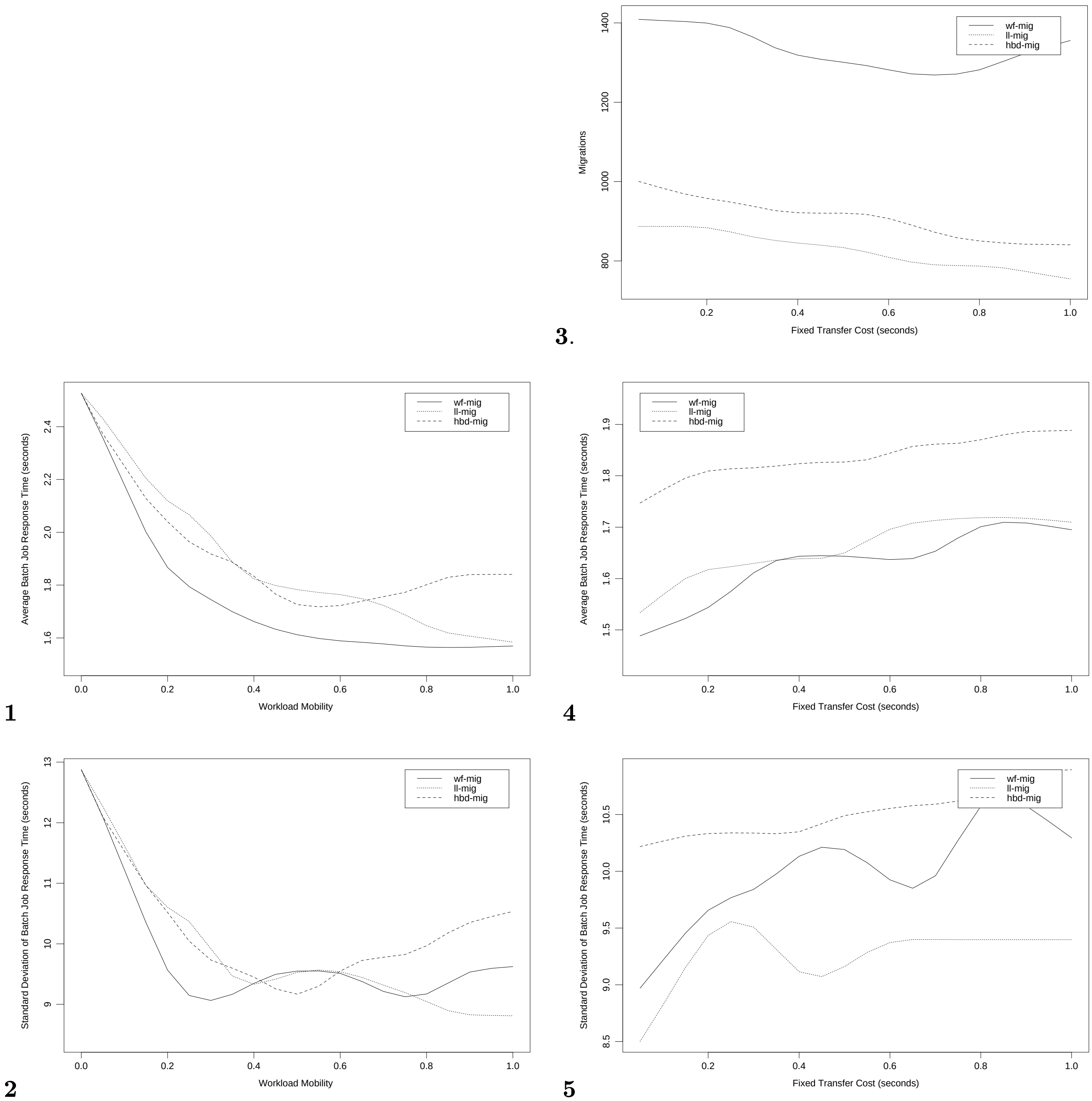


Figure C.2: *Memory bias.*

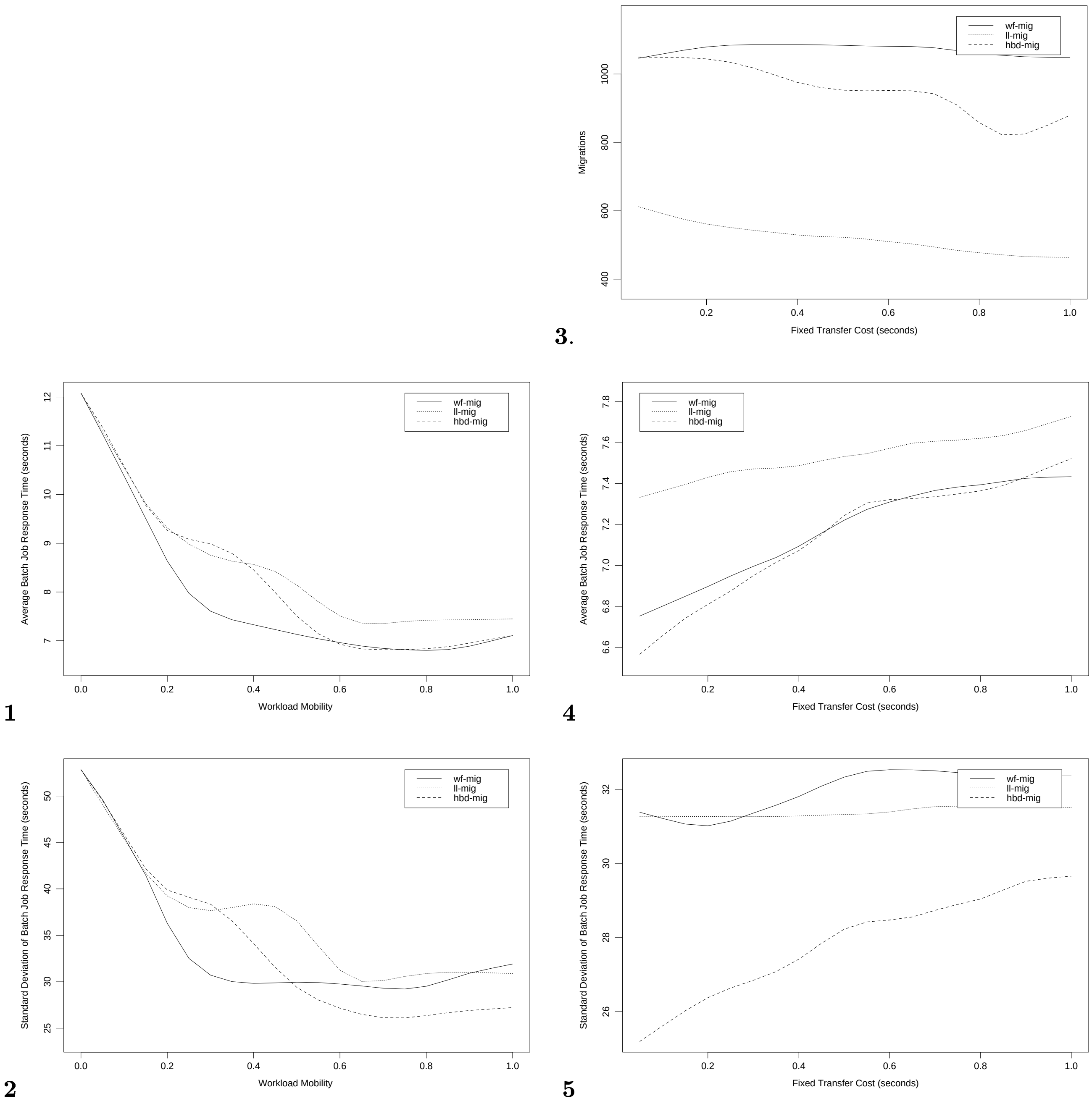


Figure C.3: *IO bias.*

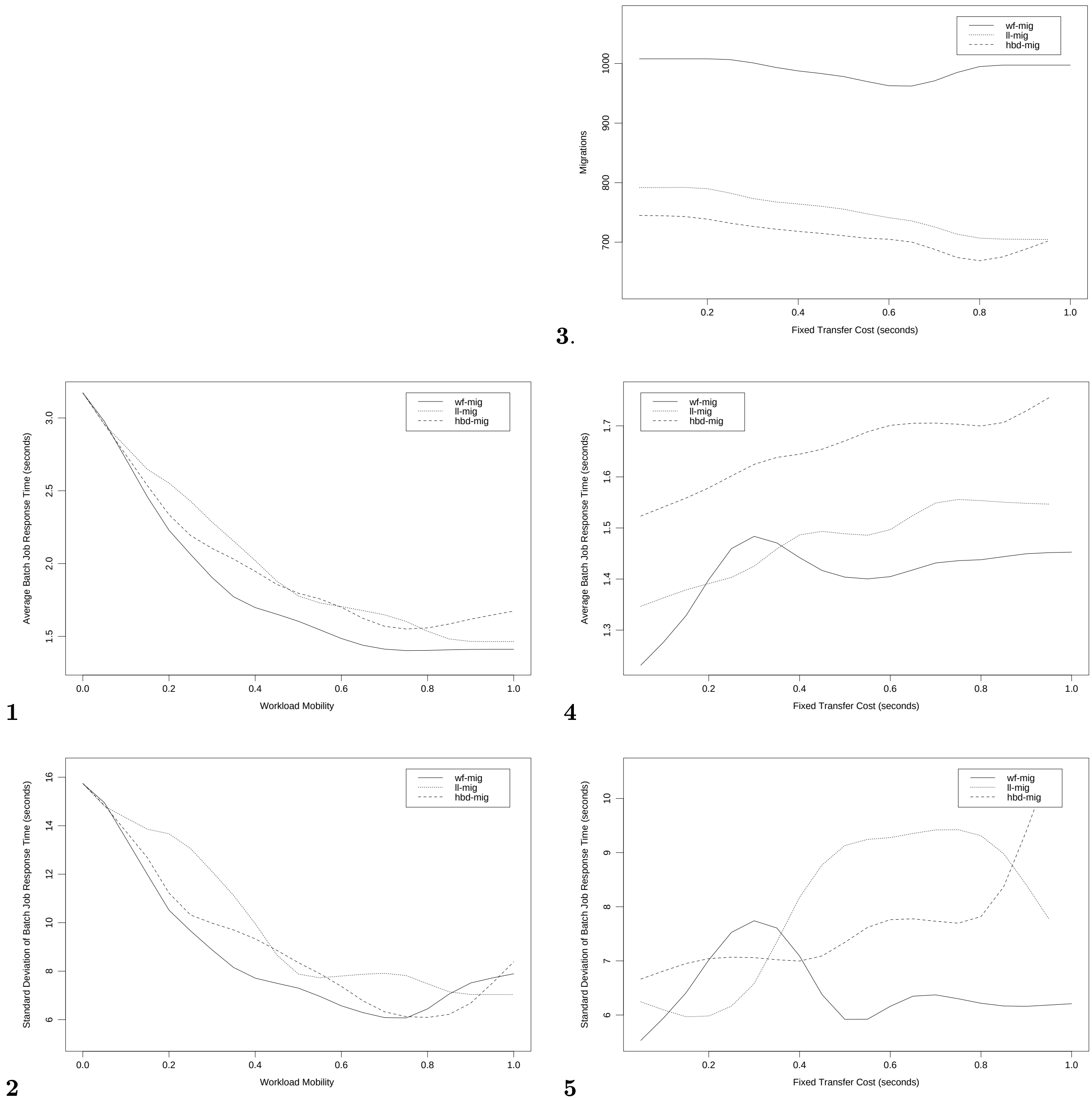
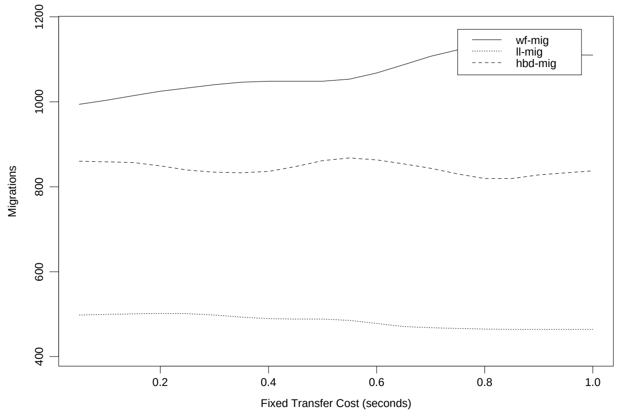
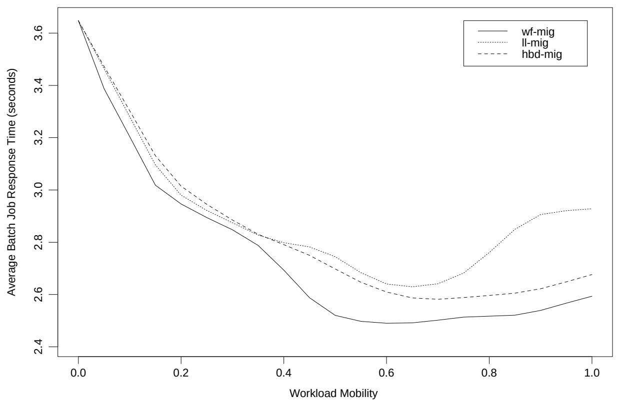


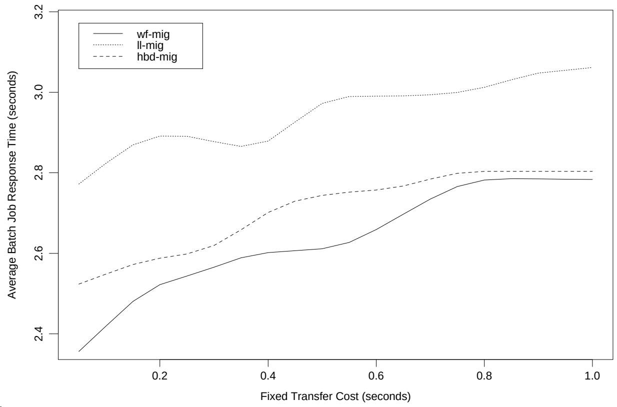
Figure C.4: *CPU and memory bias.*



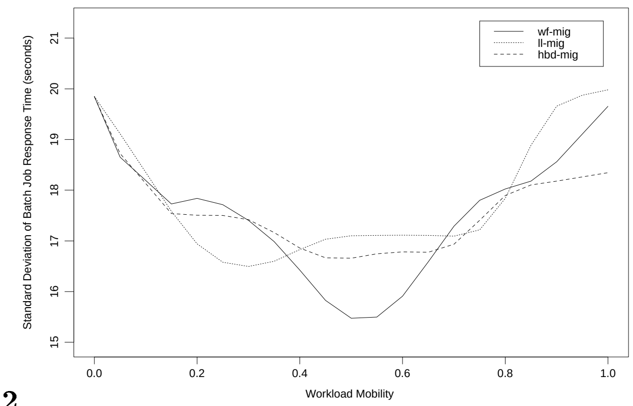
3.



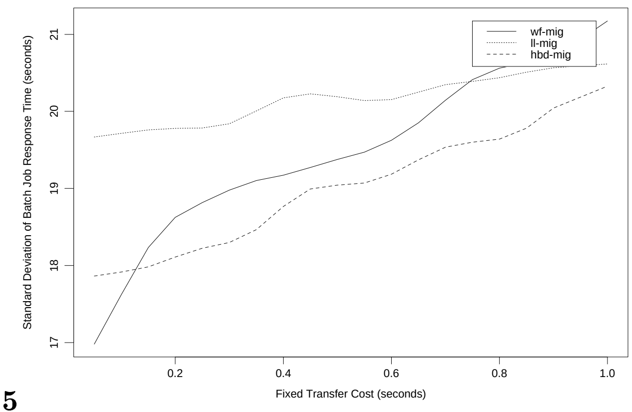
1



4

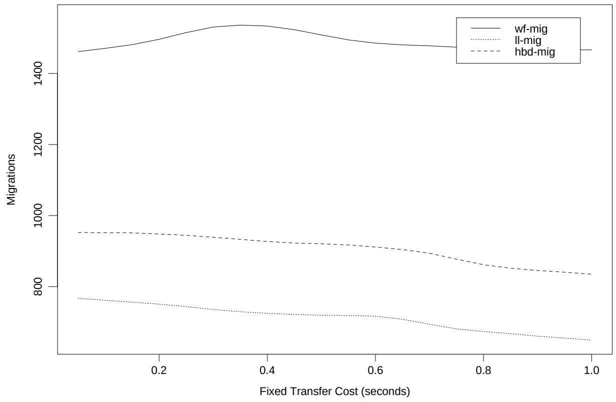


2

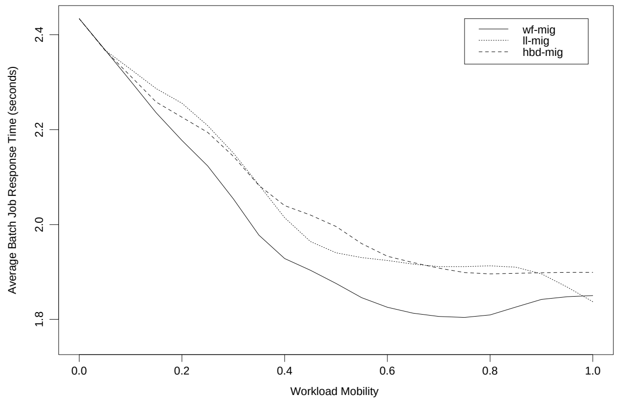


5

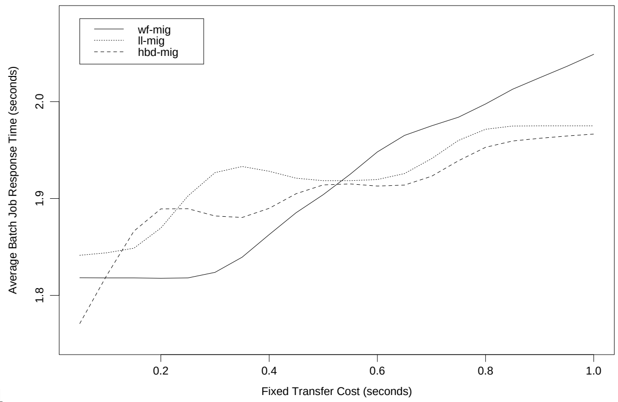
Figure C.5: *CPU and IO bias.*



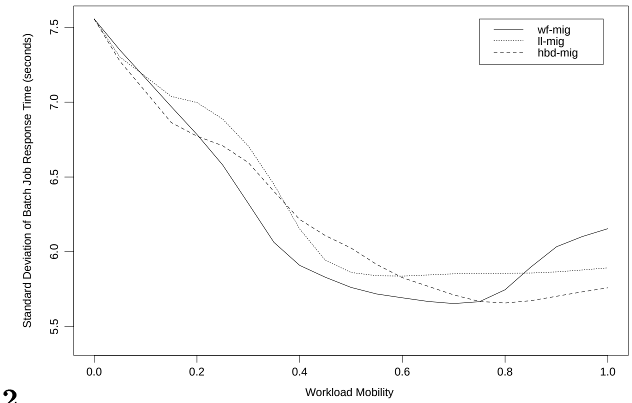
3.



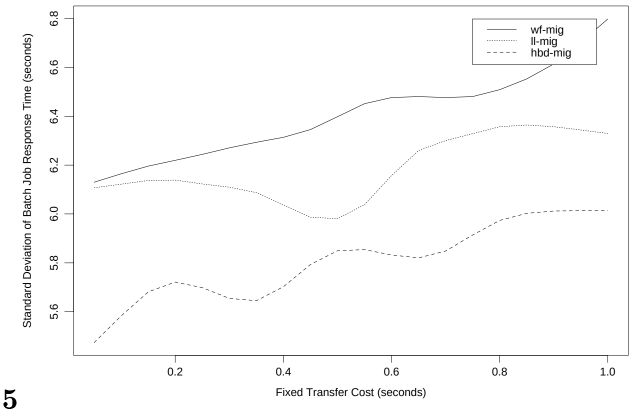
1



4

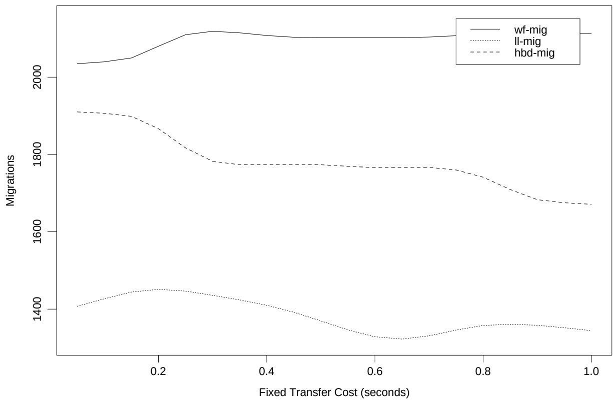


2

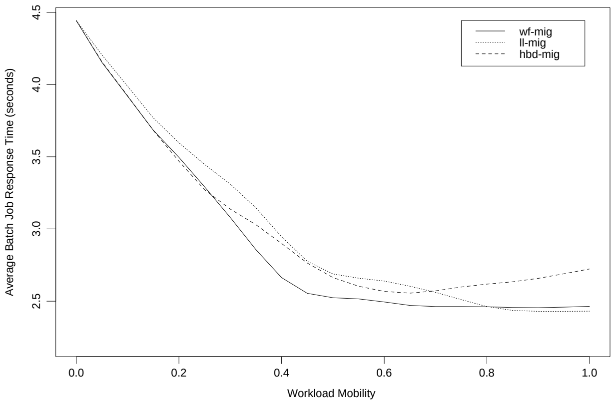


5

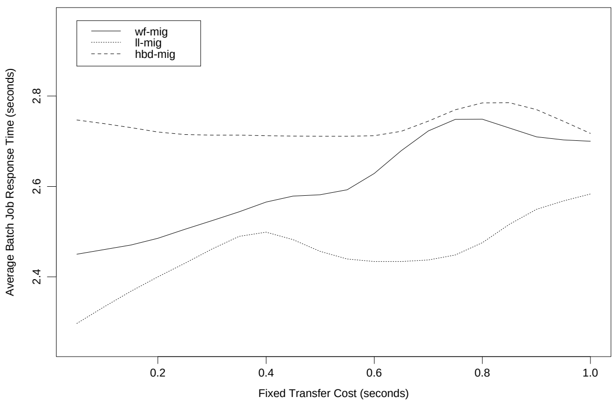
Figure C.6: *Memory and IO bias.*



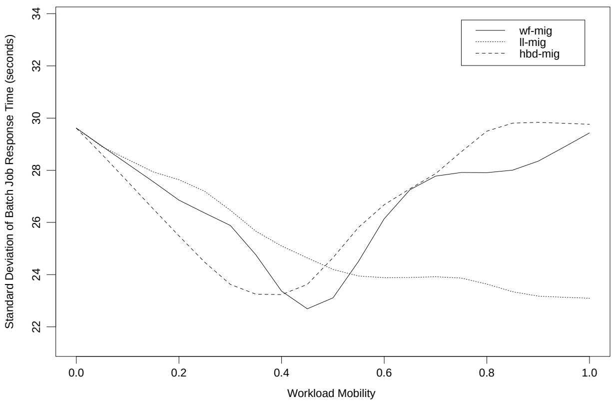
3.



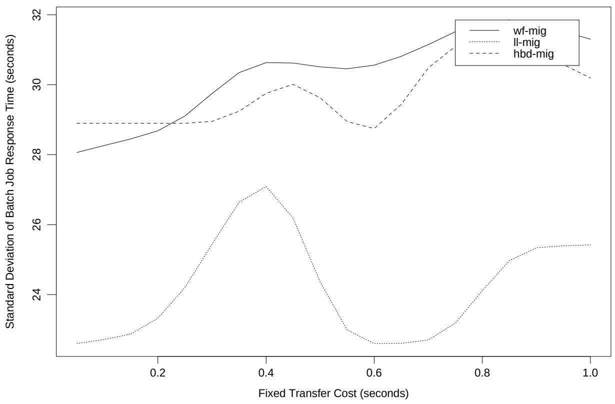
1



4



2



5

Figure C.7: *Highest load, no bias.*

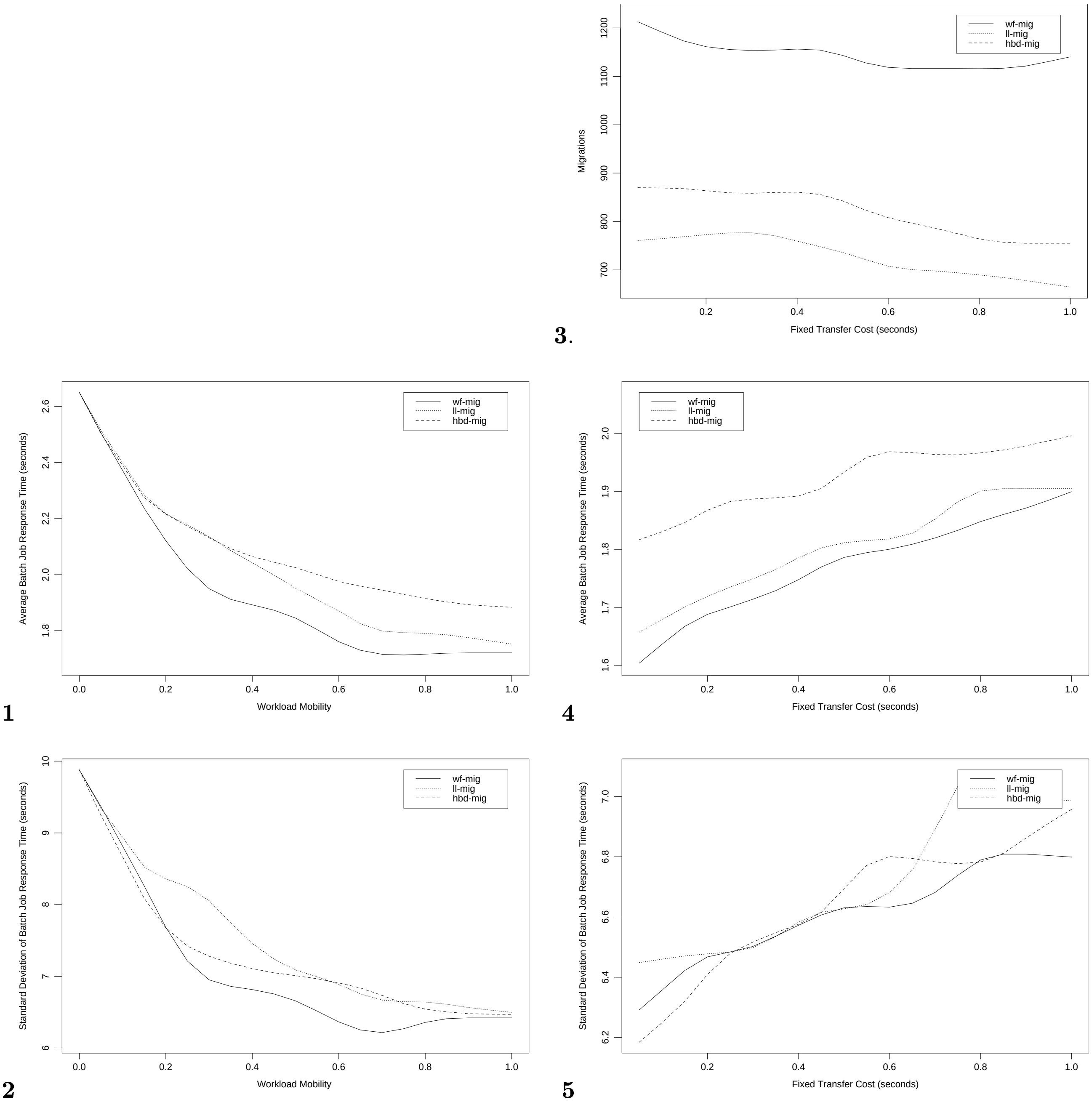
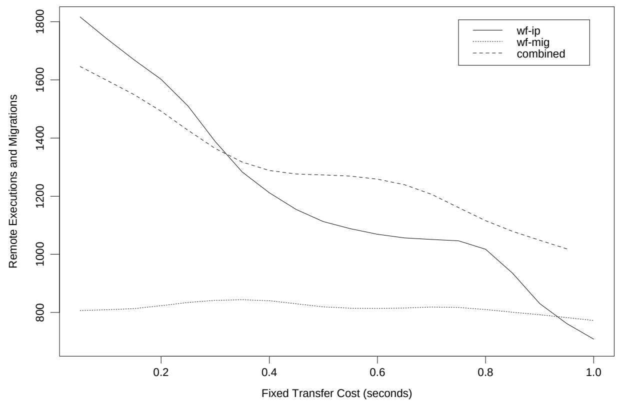


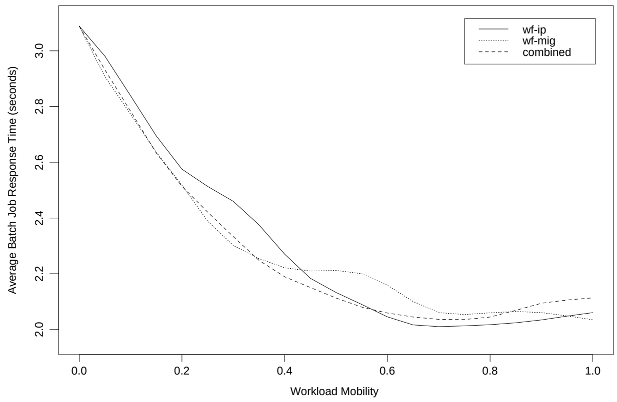
Figure C.8: *Medium load, no bias.*

Appendix D

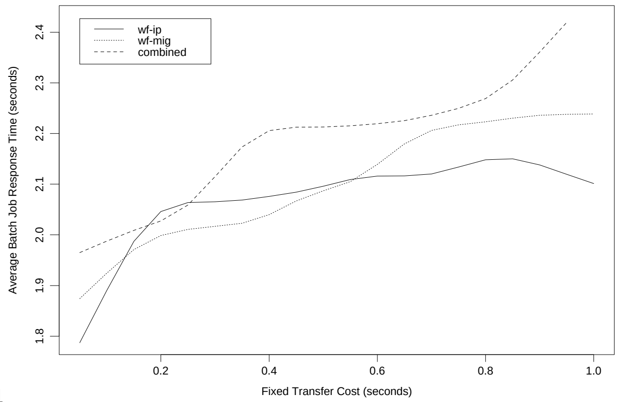
Best Compared



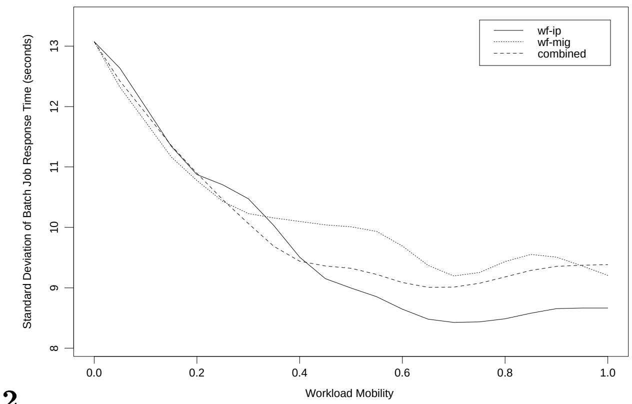
3.



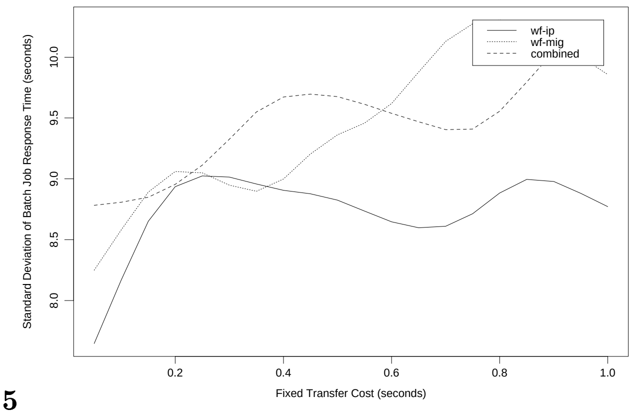
1



4

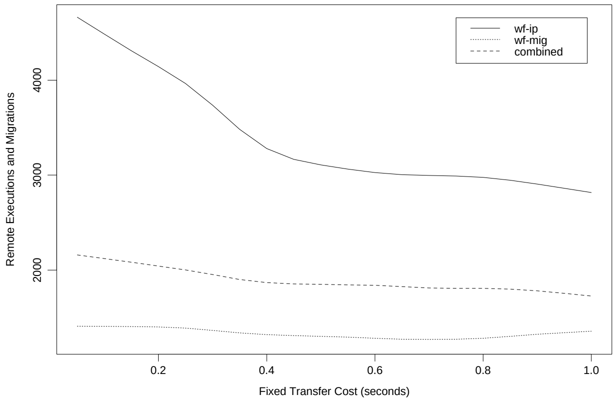


2

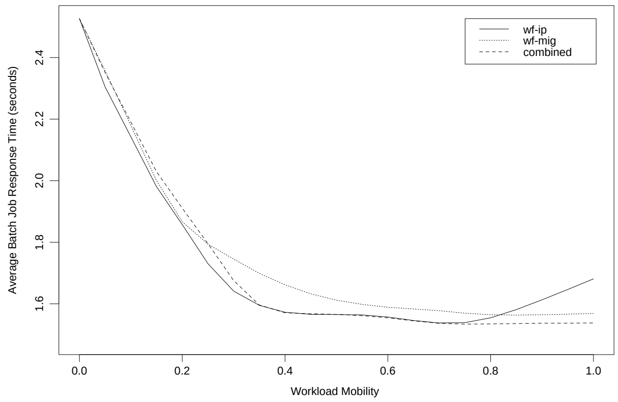


5

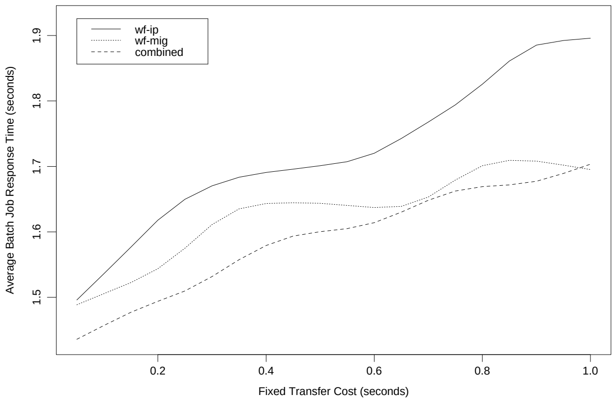
Figure D.1: *CPU bias.*



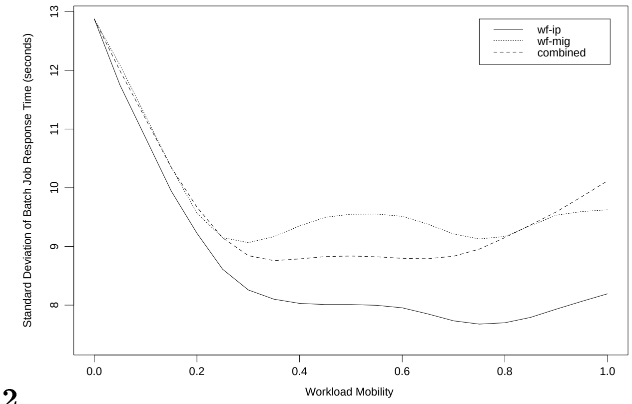
3.



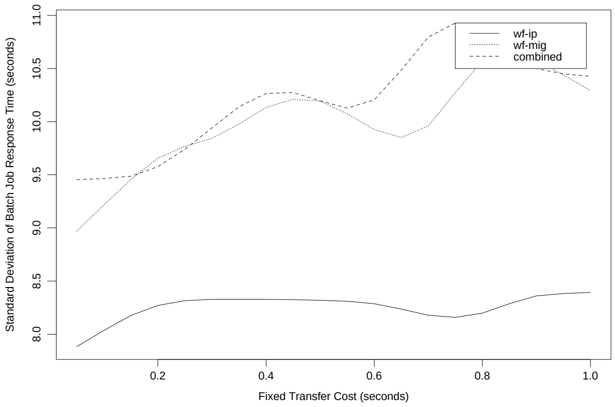
1



4

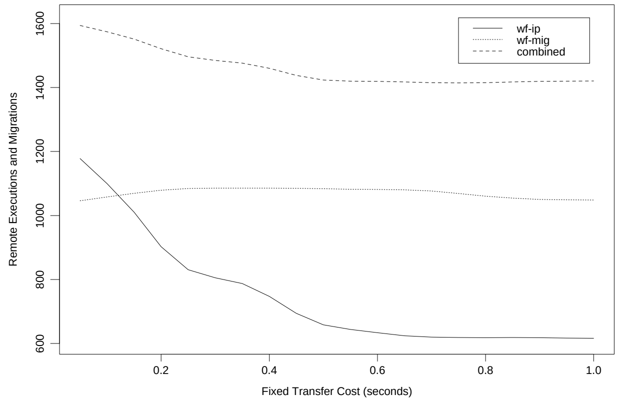


2

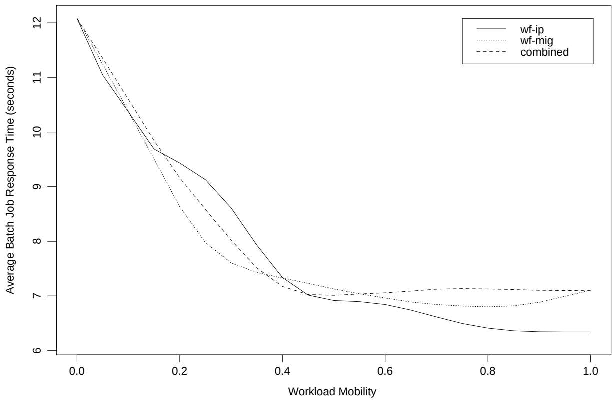


5

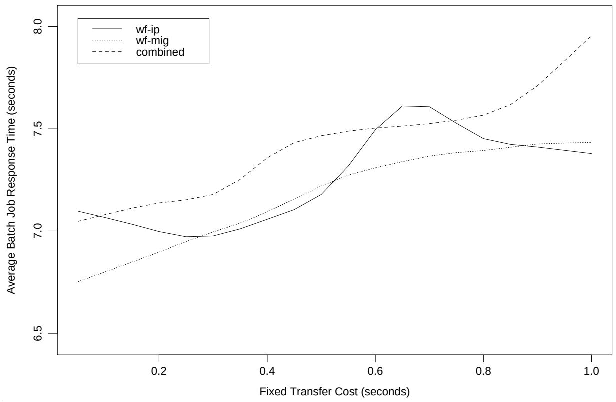
Figure D.2: *Memory bias.*



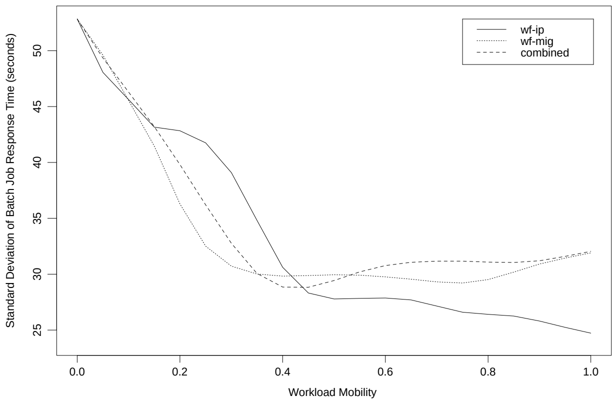
3.



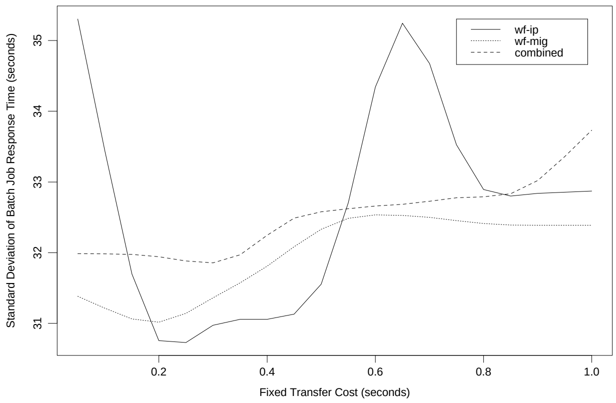
1



4



2



5

Figure D.3: *IO bias.*

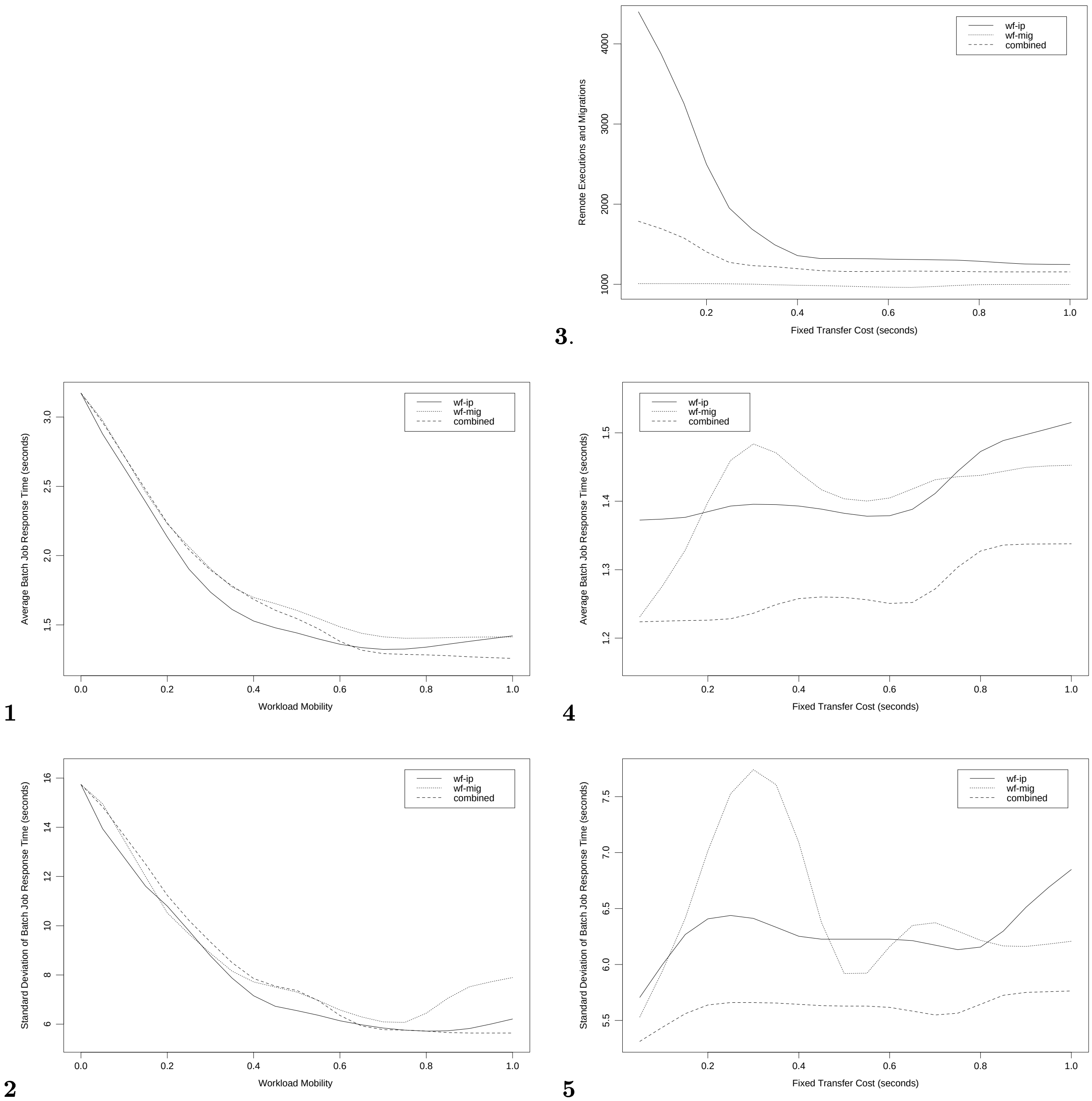


Figure D.4: *CPU and memory bias.*

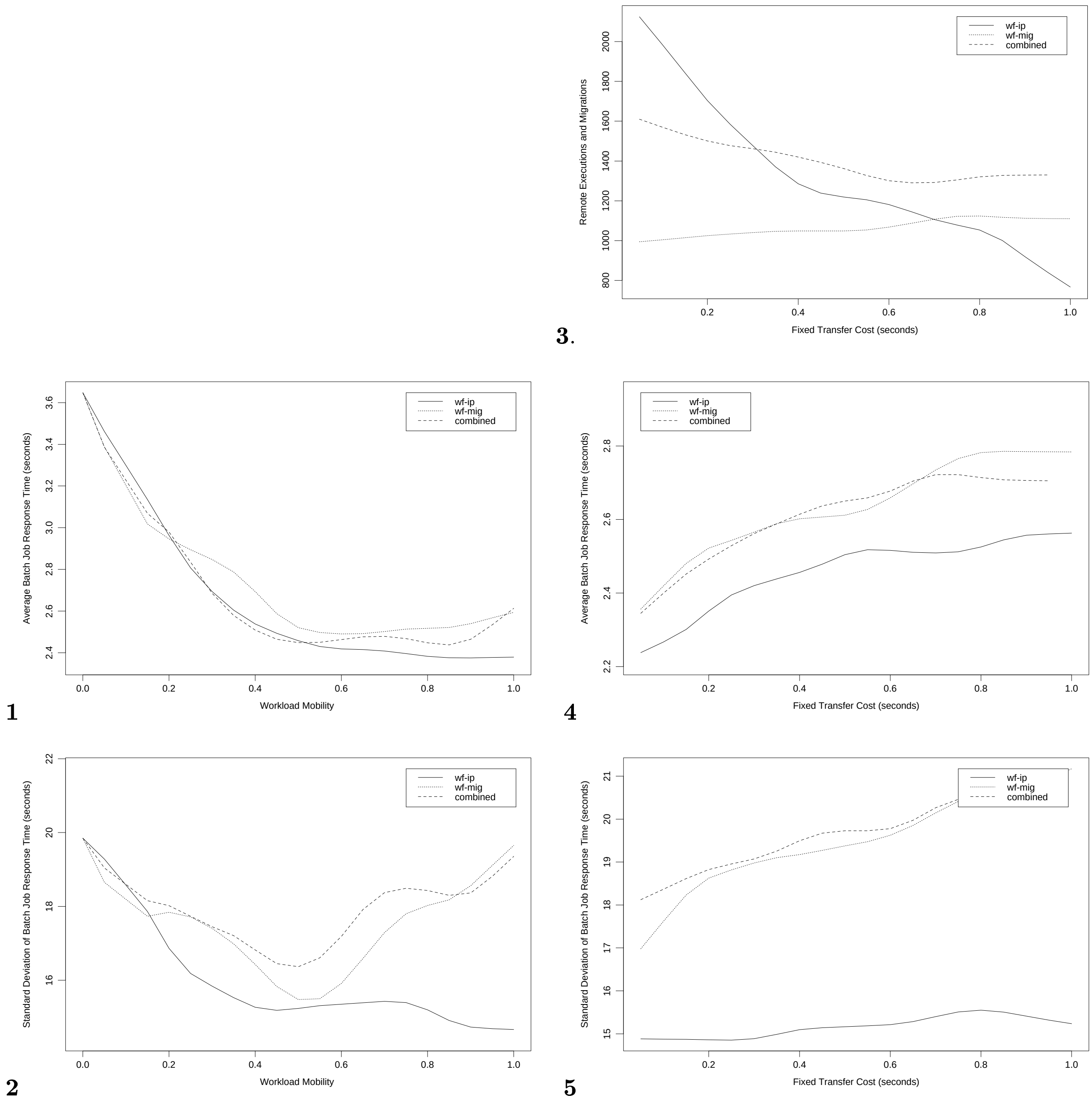
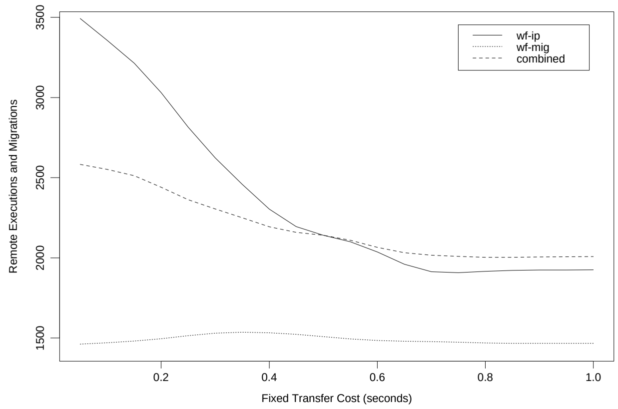
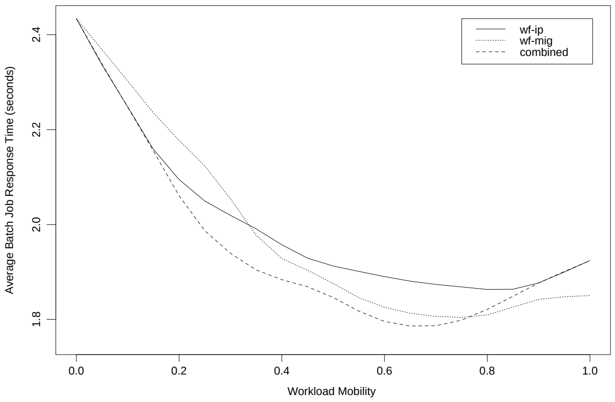


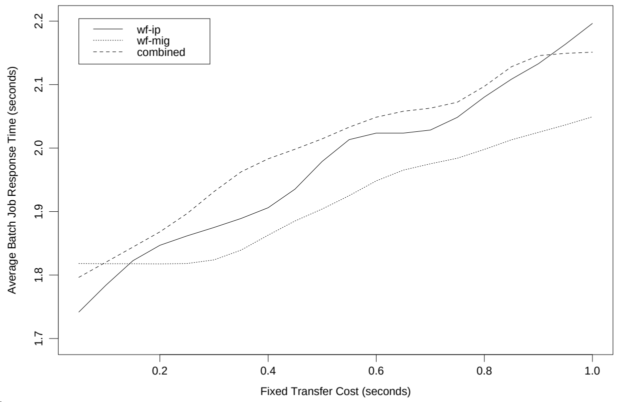
Figure D.5: *CPU and IO bias.*



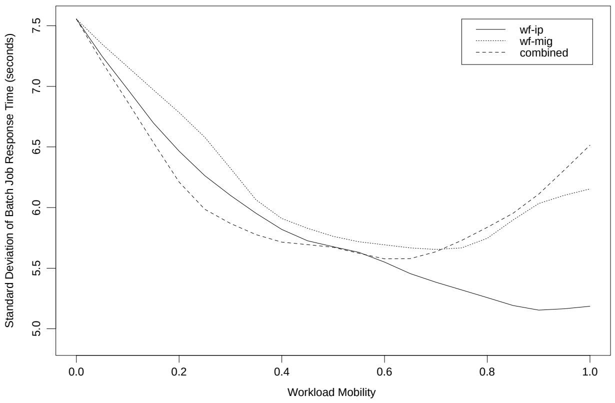
3.



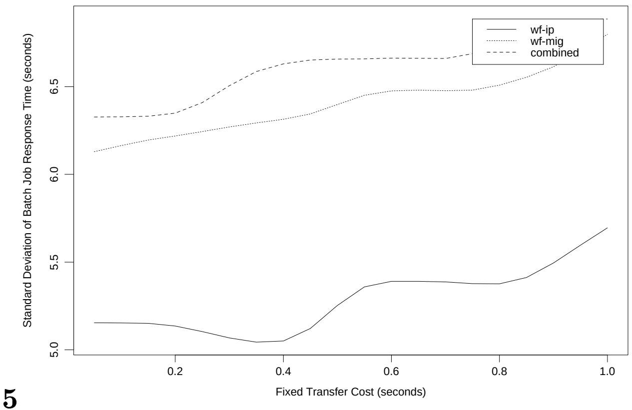
1



4

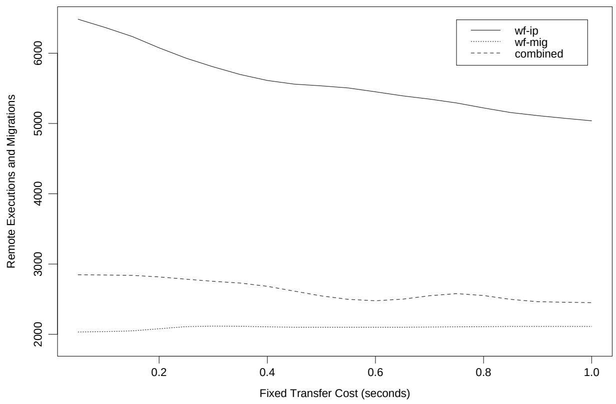


2

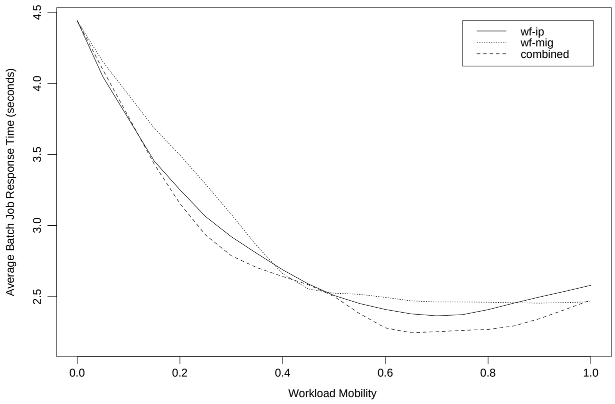


5

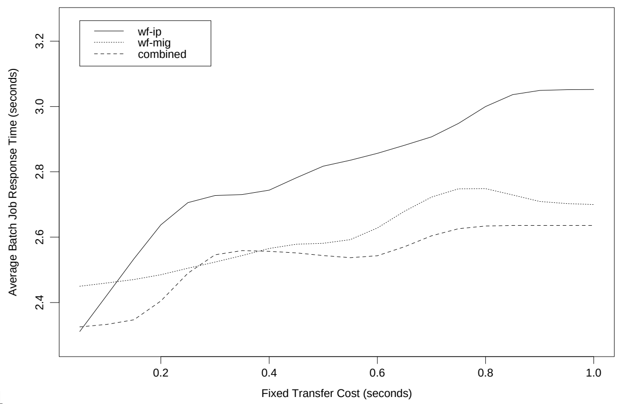
Figure D.6: *Memory and IO bias.*



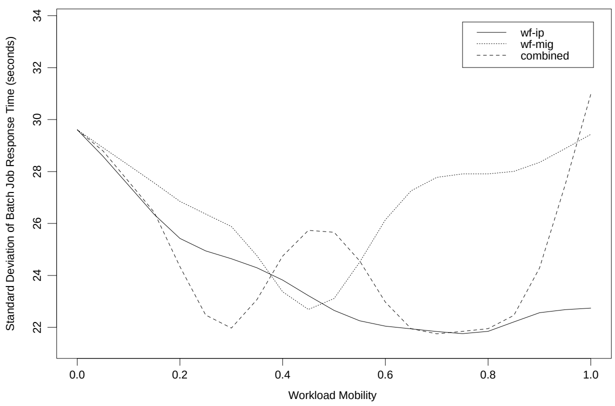
3.



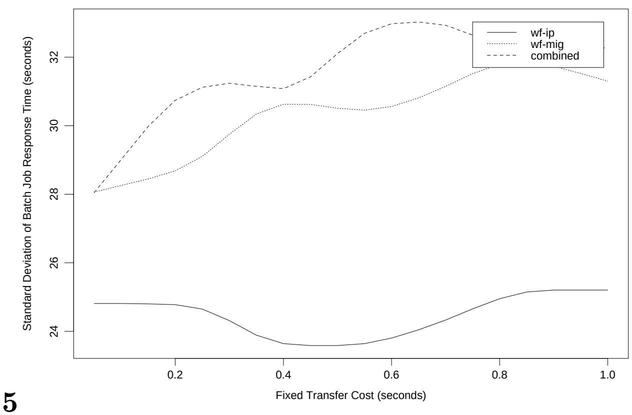
1



4

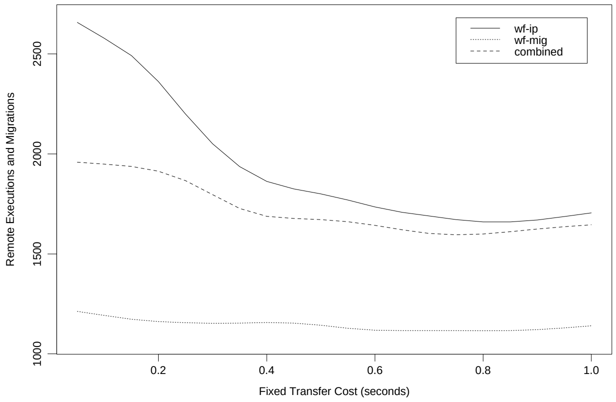


2

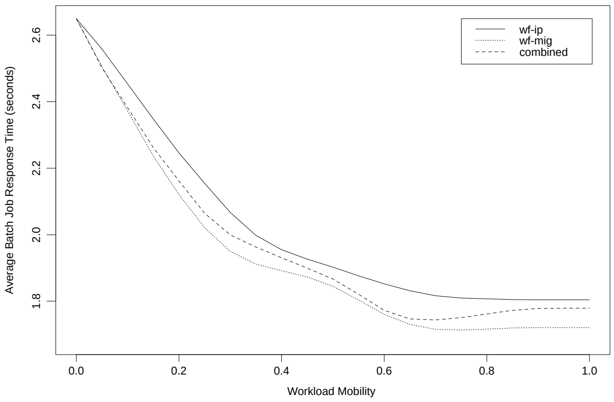


5

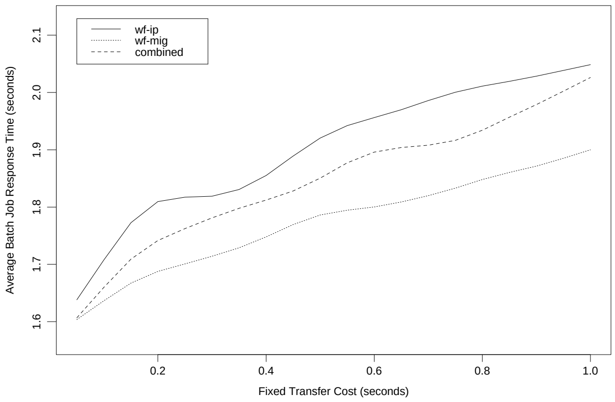
Figure D.7: *Highest load, no bias.*



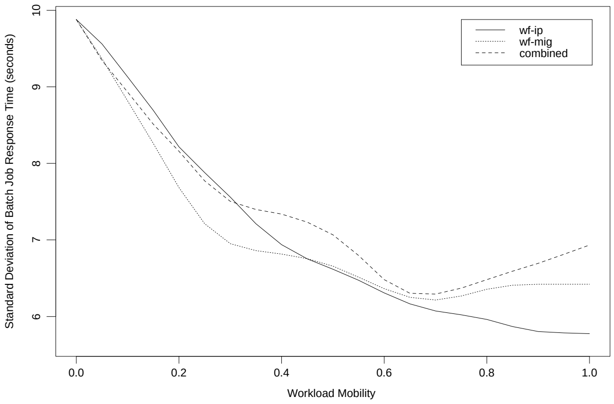
3.



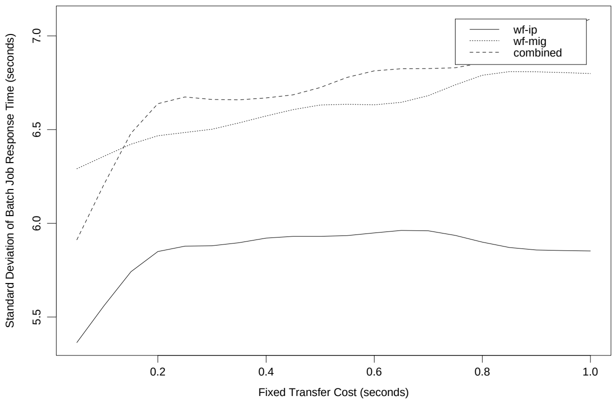
1



4



2



5

Figure D.8: *Medium load, no bias.*

References

[AF89] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, September 1989.

[AJJ+92] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent Object Migration in COOL-2. Technical Report CS/TR-92-30, Chorus Systèmes, 1992.

[BB94] Sayed Atef Banawan and Daniele Micci Barreca. A Neural Network Approach to Clustering in Workload Characterisation, 1994.

[Ber85] Brian Bershad. Load Balancing with Maitre d'. Technical Report CSD-85-276, University of California at Berkeley, December 1985.

[BLL92] Allan Bricker, Michael Litzkow, and Miron Livny. *Condor Technical Summary*. University of Wisconsin, January 1992.

[BMD94] K. Benmohammed-Mahieddine and P. M. Dew. A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems. *SIGOPS*, 28(1):66–77, January 1994.

[Bon93] Andrew Murray Bond. *Adaptive Task Allocation in a Distributed Workstation Environment*. PhD thesis, Victoria University of Wellington, 1993.

[BS85] Amnon Barak and Amnon Shiloh. A Distributed Load-Balancing Policy for a Multicomputer. *Software-Practice and Experience*, 15(9):901–913, September 1985.

[BVW95] Matt Bishop, Mark Valence, and Leonard F. Wisniewski. Process Migration for Heterogeneous Distributed Systems. Technical Report TR95-264, Dartmouth College, August 1995.

[CCG95] M. Cena, M. L. Crespo, and R. Gallard. Transparent Remote Execution LAHNOS by Means of a Neural Network Device. *SIGOPS*, 29(1):17–28, January 1995.

[CF86] Maria Clazarossa and Demenico Ferrari. A Sensitivity Study of the Clustering Approach to Workload Modelling. *Performance Evaluation*, 6:25–33, March 1986.

[CK88] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[Coc93] Adrian Cockcroft. *Sun Performance Tuning Overview.* SMCC Technical Marketing, 2550 Garcia Avenue, Moutainview, CA 94043, USA , December 1993.

[CRY94] Soumen Chakrabarti, Abhiram Rande, and Katherine Yelick. Randomised Load Balancing for Tree-structured Computation. In *Proceeding of the 1994 IEEE Scalable High Performance Computing Conference*, pages 666–673. IEEE, 1994.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[DHB95] Allen B. Downey and Mor Harchol-Balter. A note on "The Limited Performance Benefits of Migrating Active Processes for Load Sharing". Technical Report UCB/CSD-95-888, Computer Science Division, University of California, Berkeley, November 1995.

[DI89] Murthy V. Devarakonda and Ravishankar K. Iyer. Predictability of Process Reasource Usage: A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, December 1989.

[DO91] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software-Practice and Experience*, 21(8):757–785, August 1991.

[Dou90] Frederick Douglis. *Transparent Process Migration in the Sprite Operating System.* PhD thesis, University of California at Berkeley, September 1990.

[ELZ86a] D. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6:53–68, March 1986.

[ELZ86b] D. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 5:662–675, May 1986.

[ELZ88] D. Eager, E. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *Proceedings of the 1988 Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 63–72. ACM SIGMETRICS, May 1988.

[FL93] Bryan Ford and Jay Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model.* University of Utah, 1993.

[Fra77] W.R. Franta. *The Process View of Simulation.* Operating and Programming Systems Series. North-Holland, 1977.

[FZ87] Domenico Ferrari and Songnian Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. Technical Report CSD-87-353, University of California at Berkeley, January 1987.

[GDI93] Kumar K. Gosami, Murthy Devarakonda, and Ravishankar K. Iyer. Prediction-Based Dynamic Load Sharing Heuristics. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):638–648, June 1993.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[HBD95] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. Technical Report UCB/CSD-95-021, Computer Science Division, University of California, Berkeley, May 1995.

[HN90] Robert Hecht-Nielsen. *Neurocomputing*, chapter 5. Addison-Wesley Publishing Company, 1990.

[Hol96] Teijo Holzer. Performance Measurement of Task Allocation in a Distributed System. Master's thesis, Victoria Univeristy of Wellington, 1996+. Work in progess.

[JM93] C. Jacqmot and E. Milgrom. A Systematic Approach to Load Distribution Strategies for distributed systems. In *Decentralized and Distributed Systems*, September 1993.

[JXY95] Jiubin Ju, Gaochao Xu, and Kun Yang. An Intelligent Dynamic Load Balancer for Workstation Clusters. *SIGOPS*, 29(1):7–16, January 1995.

[Kar94] Mourad Kara. A Global Plan Policy for Coherent Cooperation in Distributed Dynamic Load Balancing Algorithms. Technical Report 94.21, University of Leeds, July 1994.

[KK92] Orly Kremien and Jeff Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[KL88] Phillip Krueger and Miron Livny. A Comparison of Preemptive and Non-Preemptive Load Distributing. In *10th International Conference on Distributed Computing Systems*, pages 123–130. IEEE, June 1988.

[KRK94] Thomas Koch, Geral Rohde, and Bernd Kraemer. Adaptive Load Balancing in a Distributed Environment. Technical Report SDNE-94, Fern University, Hagen, Fern University, 58084 Hagen, Germany, 1994.

[KS94] P. Krueger and N. Shivaratri. Adaptive Location Polices for Global Scheduling. *IEEE Transactions on Software Engineering*, 20(6):432–444, June 1994.

[Kun91a] Thomas Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. Technical Report TI-6/91, Institut für Theoretische Informatik, Fachbereich Informatik, Technische Hochschule Darmstadt, December 1991.

[Kun91b] Thomas Kunz. The Learning Behaviour of a Scheduler using a Stochastic Learning Automaton. Technical Report TI-7/91, Institut für Theoretische Informatik, Fachbereich Informatik, Technische Hochschule Darmstadt, December 1991.

[Leh93] Morten Lehrmann. Load Distribution in Emerald: an Experiment. Master's thesis, University of Copenhagen, June 1993.

[LJP93] Rodger Lea, Christian Jacquemot, and Eric Pillenvesse. COOL: System support for distributed object oriented programming. *Communications of the ACM, special issue on Concurrent Object Oriented Programming*, 36(9), September 1993.

[LM82] Miron Livny and Myron Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. *Proceedings of the ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[LO86] Will E. Leland and Teunis J. Ott. Load Balancing Heuristics and Process Behaviour. *Performance Evaluation Review*, pages 54–69, May 1986.

[Mar94] Paul Martin. *Adding Safe and Effective Load Balancing to Multicomputers.* PhD thesis, The University of Edinburgh, 1994.

[Nic87] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12. ACM, November 1987.

[Oss92] William Osser. Automatic Process Selection for Load Balancing. Master's thesis, University of California at Santa Cruz, June 1992.

[Pay82] James Payne. *Introduction to Simulation.* McGraw-Hill computer science series, 1982.

[PM83] Michael L. Powell and Barton P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[Raa93] Kimmo E. E. Raatikainen. Cluster Analysis and Workload Classification. *SIGMETRICS*, 20(4):24–30, May 1993.

[Rub87] Harry I. Rubin. The Design of a Load Balancing Mechanism for Distributed Computer Systems. Technical Report CSD-87-362, University of California at Berkeley, July 1987.

[Sha75] Robert E. Shannon. *Systems Simulation the art and science.* Prentice-Hall, 1975.

[SST95] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. Adaptive Load Balancing: A Study in Mulit-Agent Learning. *Journal of Artificial Intelligence Research*, 2:475–500, May 1995.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[TH91] Marvin M. Theimer and Barry Hayes. Heterogeneous Process Migration by Recompilation. In *IEEE 11th International Conference on Distributed Computing Systems*, pages 18–25. IEEE, IEEE CS Press, 1991.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33:46–63, December 1990.

[VD91] P. Vernemmen and E.H. D'Hollander. Load balancing in a distributed system using prediction and estimation. In Tzafestas, S. and Borne, P. and Grandinetti, L., editor, *Parallel and Distributed Computing in Engineering*

Systems., pages 303–308. North-Holland; Amsterdam, Netherlands, June 1991.

[WM85] Yung-Terng Wang and Robert Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[Zho87] Songnian Zhou. A Trace Driven Simulation Study of Dynamic Load Balancing. Technical Report CSD-87-305, University of California at Berkeley, September 1987.