DISTRIBUTED PREEMPTIVE PROCESS MANAGEMENT WITH CHECKPOINTING AND MIGRATION FOR A LINUX-BASED GRID OPERATING SYSTEM

HTIN PAW OO @ NUR HUSSEIN

UNIVERSITI SAINS MALAYSIA

2006

DISTRIBUTED PREEMPTIVE PROCESS MANAGEMENT WITH CHECKPOINTING AND MIGRATION FOR A LINUX-BASED GRID OPERATING SYSTEM

by

HTIN PAW OO @ NUR HUSSEIN

Thesis submitted in fulfilment of the requirements

for the degree of

Master of Science

June 2006

To my parents

ACKNOWLEDGEMENTS

I thank God for giving me the strength to persevere through all the trials of this postgraduate degree. I would like to thank my late father Dr. Mohd. Yunus for inspiring me to get into research, my mother Pn. Bibi Zahida for patiently supporting me throughout this experience, my supervisors Dr. Fazilah Haron and Dr. Chan Huah Yong for their wisdom and guidance, and also the entire faculty of the School of Computer Science in Universiti Sains Malaysia.

I would like to convey my deepest gratitude to Dr. Abdul Rahman Othman, Dr. Jim Basney, Dr. Mor Harchol-Balter, Dr. Low Heng Chin, Dr. Yen Siew Hwa, and Dr. Quah Soon Hoe for taking some of their time to shed some insight into my research problem.

I would also like to thank Josef Sipek for his help and for lending me his computer cluster, Lim Lian Tze for teaching me LaTeX, Dr. Con Kolivas for his scheduler expertise, and for their expert technical (and nontechnical) advice, Seth Arnold, Dr. Daniel Berlin, Jeff Dike, Pat Erley, Henrý Þór Baldursson, Bert Hubert, William Lee Irwin III, Christian Leber, Ahmed Masud, Zwane Mwaikambo, Eduardo Pinheiro, Rik Van Riel, Steinn Sigurðarson and the rest of my friends out there on the internet who have patiently endured my questions over the past three years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PENGURUSAN PROSES PREEMPTIF TERAGIH DENGAN PENITIKSEMAKAN DAN MIGRASI UNTUK SISTEM PENGOPERASIAN GRID BERASASKAN LINUX

ABSTRAK

Kemunculan perkomputeran grid telah membolehkan perkongsian sumber perkomputeran teragih antara peserta-peserta organisasi maya. Walau bagaimanapun, sistem pengoperasian kini tidak memberi sokongan paras rendah secukupnya untuk perlaksanaan perisian grid. Kemunculan suatu kelas sistem pengoperasian yang dipanggil sistem pengoperasian grid memberikan pengabstrakan peringkat sistem untuk sumber-sumber grid. Tesis ini mencadangkan penambahan pengurusan proses preemptif teragih kepada sistem pengoperasian GNU/Linux untuk menjadikannya sistem pengoperasian grid. Dengan menampal inti Linux dengan kemudahan penitiksemakan yang dipanggil EPCKPT, pembuktian konsep perisian tengah yang dipanggil Zinc telah dibina. Perisian Zinc menggunakan kemudahan penitiksemakan dengan cekap untuk membolehkan pengurusan proses teragih yang merangkumi penskedulan, penempatan proses grid dan migrasi proses grid. Dengan menggunakan daya pemprosesan (*throughput*) sebagai metrik pengukuran prestasi, kecekapan kemudahan migrasi proses telah diukur pada pelantar ujian grid yang terdiri daripada kluster PC di Pusat Pengajian Sains Komputer, Universiti Sains Malaysia. Proses-proses grid juga telah berjaya dimigrasikan melalui internet. Eksperimen telah dijalankan yang menunjukkan bahawa migrasi proses preemptif yang dijalankan oleh sistem pengoperasian membantu mengekalkan daya pemprosesan (*throughput*) yang tinggi tidak mengira strategi penempatan proses yang digunakan.

DISTRIBUTED PREEMPTIVE PROCESS MANAGEMENT WITH CHECKPOINTING AND MIGRATION FOR A LINUX-BASED GRID OPERATING SYSTEM

ABSTRACT

The advent of grid computing has enabled distributed computing resources to be shared amongst participants of virtual organisations. However, current operating systems do not adequately provide enough low-level facilities to accommodate grid software. There is an emerging class of operating systems called grid operating systems which provide systems-level abstractions for grid resources. This thesis proposes the addition of preemptive distributed process management to GNU/Linux, thus building a subset of the required functionality to turn GNU/Linux into a grid operating system. By patching the Linux kernel with a popular checkpointing facility called EPCKPT, a proof-of-concept grid middleware called Zinc was constructed which effectively makes use of checkpointing to provide distributed process management which encompasses scheduling, placement and migration of grid processes. By using job throughput as our performance metric, the effectiveness of the process migration facility was measured on a testbed grid which consisted of PC clusters in the School of Computer Science at Universiti Sains Malaysia. Grid processes were also successfully migrated over the internet. An experiment was carried out that showed that preemptive process migration in the operating system helps maintain system throughput that is consistently high, regardless of the process placement strategy used.

CHAPTER 1

INTRODUCTION

1.1 Concerning Grid Computing

The growing ubiquity of cheap computing power and high-speed networks have given birth to distributed computing, which combine the resources of networked computers and harness the resulting combined power of its constituent computing elements. Tanenbaum and Van Steen [74] describe distributed systems as "a collection of independent computers that appears to its users as a single coherent system". From this definition, it could be inferred that a distributed system has a generic goal of providing a transparent and coherent service to users of systems comprising more than one physical computing machine. It could be said that grid computing is a special instance of distributed systems. Grid technology allows us to collectively perform complex computational tasks that would not be feasible on a single computer by means of pooling together resources that are shared by various institutions, organisations and individuals. Foster and Kesselman define computational grids as "hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities"[22].

Grid computing grew out of metacomputing, an early effort to consolidate disparate and diverse computing resources to take advantage of the resulting combined computing power. Previously, a user trying to utilise such a wide collection of different resources had to put up with manually configuring and scheduling jobs on different user accounts, machines and programs. Current developments have produced automated tools and advanced job

scheduling and monitoring technologies to assist the user in the sharing of this collection of resources. These technologies form the fabric of grid computing.

Various approaches have been taken to designing software that control and facilitate the computational grid. Usually, grid software is implemented as middleware, a layer of abstraction that lies between user programs and the host computational hardware and software. Examples of such software are Globus [21], Condor [46] and Legion [27]. These systems provide a collection of services for both users and user programs to help aggregate and share computing resources.

This chapter is a prelude to the design and implementation of Zinc, a layer of grid software for the GNU/Linux operating system developed with the idea of making grid middleware as transparent and as easy to use as possible, while maximising the job throughput of the grid. The design of Zinc is from a perspective of an operating systems programmer, while its implementation covers kernel modifications and userspace tools to support those modifications. The primary goal of the system is to provide a foundation for which we can experiment with the possibilities opened up by extending the operating system to accommodate grid computing, with regard to process management. These extensions can be seen as a first step towards the creation of a *grid operating system*, which is an operating system that provides an abstraction of grid services to make the technology more transparent and easy to use by both end-users and programmers.

1.2 Research Motivation

1.2.1 Investigating The Factors That Influence Job Throughput

Much literature has been written about the effect of process migration, scheduling algorithms and other aspects of distributed computing on high-performance problems. How-

ever, research on high throughput computing have not been as extensive, and warrants further investigation. We believe that the grid's primary function is an enabler of high-throughput computing. Although many or most hardware in the grid is going to provide high-performance computing facilities to its users, the entire system as a whole exists to maximise the amount of work done with the resources available. Hence, looking into what conditions are favourable to increase the throughput in our distributed system is justified, and the results of our observations can be used to build better grids.

1.2.2 Introducing Distributed Process Management Support Into GNU/Linux

GNU/Linux[1] is a Unix-like operating system which is worked on by various programmers over the world, both voluntarily and or for commercial purposes sponsored by various companies. GNU/Linux is a collection of open source programs that make up a free operating system which can be modified and redistributed by anyone. At the heart of GNU/Linux is the Linux kernel, a free operating system kernel licensed under the GNU General Public License (GPL). Linux was initiated by Finnish programmer Linus Torvalds, and at the time of writing, he continues to spearhead its development in collaboration with thousands of developers world wide to further improve and enhance the Linux kernel. The userspace of GNU/Linux consists largely of utilities derived from the GNU project founded by MIT hacker Richard Stallman to create a truly free Unix-like operating system. Since GNU/Linux tries to be a clone of Unix, it is also a centralised network-enabled operating system by design. We have thus chosen GNU/Linux as a platform for our grid operating system research, so that it can be extended to facilitate grid-specific requirements and

[1] In print, the usage of both the terms "GNU/Linux" and "Linux" refer to the operating system based on the Linux kernel. There is a difference of opinion on whether or not the "GNU" part should be included when referring to the OS, but for the purpose of this thesis, the distinction between GNU/Linux and Linux is that GNU/Linux refers to the complete operating system (with userspace, C libraries, compilers and all) whereas Linux refers to just the Linux kernel.

investigate process migration.

Most of the tools available in GNU/Linux distributions are clones of the original Unix tools, or new software developed from scratch to make a functional desktop and server. However, neither the Linux kernel nor the userspace of GNU/Linux is designed with grid extensions in mind. There are a number of kernel-related projects which provide checkpointing and facilities for process migration such as MOSIX [3], but have not been fully developed for the purpose of internet-based migration for grid computing. MOSIX assumes a persistent, reliable and high-bandwidth network connection is available between hosts in the distributed system, an assumption which we can not make for internet migration.

Most grid software such as Globus [21] or Condor [46] have GNU/Linux versions, but few projects attempt to fully integrate the userspace tools with added functionality in the operating system. Moreover, most distributed operating system projects were initiated before grids became popular, thus there is little effort to support grid computing in research distributed operating systems. However, the Plan 9 [53] operating system is quite well-suited to grid computing, because of it's unique resource abstraction mechanism that presents everything on the system as network accessible files, even CPUs and other devices. However, Plan 9 does not support process migration or dynamic load sharing. Also, the problem with using research operating systems is that there is very little hardware and software support for them (most do not implement the full feature set of modern Unix-like systems), and all applications that want to take advantage of the system must be at least recompiled, if not rewritten (plus there is inertia when users need to switch operating systems).

Therefore, our motivation in choosing GNU/Linux as our vessel for investigating the issues surrounding distributed process management, wide-area process migration, and grid

operating system design is because of the following:

1. Non-restrictive licensing terms for copying, modifying and redistributing the system.
2. Customisable open source kernel, and userspace software.
3. A wide range of free developer tools plus support for almost all major programming languages.
4. Very popular and supports a wide range of hardware device drivers.
5. Popular computational, scientific and grid software is available for it. These existing tools could benefit from additional grid-specific improvements.
6. It is evolving rapidly. Every few months, a new Linux kernel is released, with more feature added with each release. This rapid development process gives ample opportunity for new functionality to be included into a popular operating system (at least, gradually). This helps overcome the inertia of organisations and users refusing to totally change their operating systems, a problem described by the Legion team in Grimshaw *et al.* [26].

While GNU/Linux has been used extensively in grid computing, it was not designed as a grid operating system from the ground up. Padala [58] has proposed some enhancements to the network stack to the Linux kernel for improving network performance for grid applications. However, there have yet been no attempts to add on grid functionality to to GNU/Linux at a more fundamental operating system design level. Our research explores the idea of what a grid OS should look like, and proposes the design of a system for distributed process management as an enhancement to the GNU/Linux operating system.

1.2.3 Reducing Cruft

The New Hacker's Dictionary [63] defines "cruft" as excess, superfluous junk; used especially of redundant or superseded code. Crufty software is software with a design that's overly (and perhaps unnecessarily) complex. The design philosophy of the Globus toolkit is to work at a middleware layer, using only internet protocols. The justification for this decision as presented in [23] is to enable Globus to work on heterogeneous architectures and operating systems, and that "traditional transparencies are unobtainable" for grids. However the introduction of various new APIs in each of the components of Globus also increases the complexity of utilising a grid.

We disagree with Foster and Kesselman that trading off transparency and simplicity for heterogeneity is a necessary compromise in creating a grid. In Gabriel's essay on the design of Lisp [24], he characterises two software design strategies, one called "The MIT Approach" and another called "Worse-Is-Better". Both approaches stress the simplicity of design, where the "MIT Approach" would try and do the "right thing", where simplicity of the interface is more important than the simplicity of the implementation, whereas the "Worse-Is-Better" philosophy it is the other way around. The Globus toolkit however is both complex in terms of interface and implementation, which is in sharp disagreement with both software design philosophies. The simplicity of the design philosophy of Unix influenced the proposal of the implementation of a grid operating system in this thesis.

We also assert that it is possible to provide extra transparency via operating systems modifications while maintaining the same amount of support for heterogeneous platforms as Globus does now. Since operating system extensions are mostly transparent to userspace, it is possible for toolkits such as Globus to make use of the underlying grid features when available and still achieve its goals. The advantage of operating system support for grid

functionality however, is that given a sufficiently large collection of computers of the same architecture, it is possible to create a grid without complex middleware toolkits. Considering that the Intel x86 architecture continues to be the most prevalent computing platform, plus the growing popularity of the GNU/Linux operating system, it is not inconceivable that a computational grid of reasonable size and usefulness can be constructed with relatively homogeneous hardware.

1.2.4 Enabling Internet Computing

There exists a vast pool of computing resources worldwide, and the advent of fast internet technologies have enabled organisations willing to share their computing facilities to do so at an unprecedented level. Currently, projects such as SETI@Home [1] and Folding@Home [68] create a high-throughput computational environment via specialised programs to perform their tasks. We hope that with a grid operating system, it will be possible to easily create generic programs that work like SETI@home and Folding@Home.

1.3 Research Objectives

1.3.1 Integrating Process Checkpointing And Grid Process Scheduling Into The Linux Kernel

Currently, the Linux kernel does not support process checkpointing, a feature necessary for process migration to work. Therefore, we will port and update the EPCKPT [61] checkpointing patch into the Linux kernel 2.4.22. We will also tweak the default kernel process scheduler to better handle CPU-intensive processes by enforcing a policy that favours long-running processes and by allowing userspace to have better control over process priorities. This will allow us the necessary functionality to create a foundation for our process migration and grid process management research.

1.3.2 Creating A Prototype Grid Process Management System

With the necessary modifications to the Linux kernel, we will thus build a proof-of-concept grid process management software in userspace called Zinc. It will incorporate a userspace scheduler, monitoring daemons and command line tools for the user to submit jobs. The design goals for the system are as follows:

1. Transparency – the user must be able to submit regular programs as jobs in the grid without modification
2. High throughput – the system will try to accomplish the most amount of work for as long as it runs
3. Adaptability to dynamic resources – the system will adapt to the variable conditions of the grid
4. Decentralisation – the system must reflect the decentralised nature of the grid
5. Minimising residual dependencies – the system must try to minimise the residual dependencies of a process when migrating it

With this prototype system, we will have a controlled environment that will enable us to perform further experiments on throughput in a grid operating system.

1.3.3 Enabling Wide-Area Process Migration

For the paradigm of "grid processes" to be complete, we must allow processes to migrate around the grid to any node connected to the system. This requires that processes be able to migrate over wide areas, as the grid is a large-scale distributed system that may even span continents. Therefore, we will determine whether grid process migration is feasible

over a wide area by conducting an experiment to see the time required to migrate grid processes over the internet across continents.

1.3.4 Investigating The Factors Influencing Throughput In The Grid OS

We are interested in the question of whether or not preemptive process migration will help job throughput in the grid operating system, and also the factors that influence throughput. Since grid operating systems are still in their infancy, we will use the prototype that we develop to conduct our experiments, as we will be able to control external factors while implementing only the features that we need.

1.4 Scope Of Research

In defining the extensions to operating systems (in this case, Linux) for grid computing, substantial changes need to be done to all the subsystems of the OS. However, for the purpose of this thesis, we will restrict the scope of the implementation to distributed process management and process migration on the grid for the purpose of experimentation and implementation of a prototype kernel. Thus, grid filesystems, I/O, distributed device management, and distributed memory management was not implemented. These topics however, are discussed briefly in the last chapter. The implementation of process migration assumes the processes will not be performing inter-process communication. Thus the processes are "atomic" and may move about freely independent of other processes. The last chapter also discusses a scenario where IPC is allowed between processes and how wide-area process migration may take place in such a situation.

In the implementation of the Zinc grid process management framework, our goal is to create a proof-of-concept system to provide for us a controlled experimental test-bed

to test the feasibility of wide-area process migration and to investigate the factors that influence throughput in the grid OS. Therefore, no benchmarking will be done to compare Zinc with existing similar systems such as Condor or MOSIX, as the latter projects have different design goals, thus a meaningful benchmark is not possible without compromising our own goals.

1.5 Contributions

This thesis explores the outcomes of adding explicit features to support grid computing into the Linux operating system. The primary contribution of this research is the introduction of the concept of grid process management and global process migration via the internet to GNU/Linux. Grid process management is a subset of the functionality required for a grid operating system, and is the subset that was chosen as a focus for this thesis. The design issues with Linux that need to be addressed when extending the operating system for grid process management were identified. The goal of the grid process management implementation is to provide transparent, wide-area process migration and a means to manage the aforementioned processes. To this end the EPCKPT [61] checkpointing patch available on the internet was applied to the Linux kernel as a foundation for the distributed process management algorithms.

A two-level scheduling system was designed for the grid operating system which consists of a modified kernel scheduler which was produced in collaboration with Linux kernel developers [32], and a userspace distributed process scheduler which was implemented in a program called Zinc. Zinc is a prototype proof-of-concept implementation of resource discovery, process scheduling and execution monitoring software that takes advantage of the checkpointing mechanism and the kernel scheduler in the modified Linux kernel.

Within the Zinc userspace scheduler, an algorithm called Zinctask was introduced for placement of jobs on a distributed system which makes decisions based on state information collected from all nodes in the system. This algorithm improves upon placement algorithms based on run queue length alone [19] by adding information on full distributed state via the Name Dropper resource discovery algorithm. The Zinctask algorithm also takes into account the staleness of state information when making decisions, as well as memory and CPU loads of the system. Zinctask is evaluated against random placement of processes and is found to be superior to it in almost all scenarios in the grid test-bed used for the experiments. Together with Zinctask and process migration, the Zinc-enabled Linux-based grid operating system yields both high throughput and creates an efficient load-distribution system.

Finally, with the implemented prototype systems and grid test-bed, the factors influencing the throughput of grid computing jobs were studied. The factors of interest are:

1. The availability of preemptive processes migration.
2. The placement strategy of processes.
3. The configuration of the machines in the grid.
4. The length of the majority of jobs that are submitted to the grid.

It was discovered that the different interactions between these factors influence throughput on the grid test-bed, and certain combination of factors produce different levels of throughput.

1.6 Summary And Organisation Of Thesis

This research aims to bridge the gap between existing grid middleware and operating systems development both of which are currently not integrating in a way to provide transparency to the user. Our goal of unifying these domains is presented in figure 1.1.

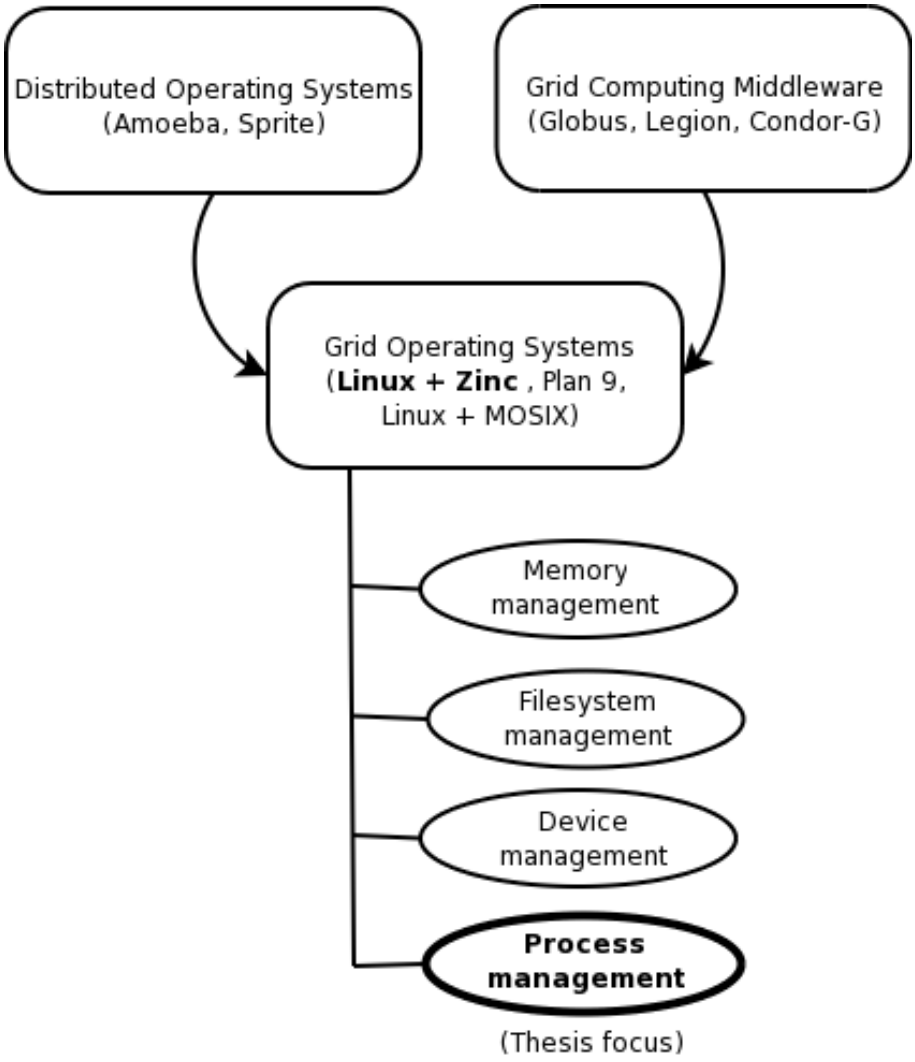

Figure 1.1: The Unification Of Operating Systems And Grid Computing

The rest of this thesis is organised as follows. Chapter 2 presents a survey and discussion of existing research related to our own. Chapter 3 gives a brief look at the design goals we have with our Zinc system, whereas the details of the design is found in Chapter 4. Chapter 5 describes the implementation of all the components of Zinc in depth. The experiments

we carried out with our system and grid test-bed is presented in Chapter 6, together with the results and discussion. Chapter 7 provides a conclusion and summary of the research plus suggestions for future work.

CHAPTER 2

RELATED WORK

2.1 Introduction

In this chapter, we discuss the literature on existing work that is relevant to our research. Firstly, the properties of the grid environment is discussed in section 2.2. Then, in section 2.3 we try and define what a grid operating system is based on previous definitions of existing operating systems. Section 2.4 discusses an efficient resource discovery algorithm that we use for Zinc. Section 2.5 presents definition of grid scheduling while section 2.6 discusses distributed scheduling algorithms design choices. Next, section 2.7 surveys process migration techniques while section 2.8 discusses some existing systems which implement process migration in different ways. The emerging field of grid operating systems are discussed in 2.9 while a summary of the chapter is provided in 2.10.

2.2 The Grid Environment

In general, the computational grid comprises the following:

1. A set of resources which are shared to the users of the grid. Resources can mean any computational infrastructure, such as hardware like CPU, RAM, disk space, network bandwidth and software like databases, shared libraries and compilers.

2. Middleware to facilitate the coordinated sharing of all these resources to all the users. Grid software will automate the authentication of users, allocation of resources,

execution and monitoring of jobs, maintain the quality-of-service, throughput, and security of the entire system.

Grid computing shares some similarities with cluster computing. They both take advantage of the abundance of cheap hardware, and to some extent perform complex computational tasks in a collaborative manner between the different processing elements. However, there are a few differences between the two technologies:

1. Clusters are tightly coupled, with processing elements consisting of individual computers connected to each other via a high-speed networking interconnect such as fast Ethernet, Gigabit Ethernet or a specialised interconnect technology such as Myrinet. Grids are usually built on a bigger scale, encompassing distributed systems networked over wide distances such as LANs, WANs and the internet. The individual processing elements of a grid can be individual computers, mainframes or entire clusters.

2. Clusters are centrally administered, and its processing elements are physically located close to each other, usually in the same room. For grids, each processing element may be independently administered by different parties, and each component that comprises the grid may be located at geographically distant locations.

3. Clusters usually consist of homogeneous processing elements. Each node in a cluster (with perhaps the exception of the master node) have identical architecture, the same hardware and software configurations and usually cluster administrators try to set up a single system image with their clusters. Grids are usually comprised of different types of computers, storage devices, instruments and other networked gadgets, creating a heterogeneous computing environment for each of these machines and devices will have its own architecture, operating system, system libraries, and other features unique to each machine or device.

With these differences, there come certain implications that make software and algorithms suited for cluster-type operation unusable or inefficient on grid systems. The following factors have to be taken into account when constructing grid software:

1. The bandwidth and low latency readily available for cluster communication is not guaranteed on a grid. The more widely distributed the components on the grid, the more prone it is to suffer from bandwidth congestion, high latency and lag times, and other undesirable effects of wide area networking.

2. The possibility that processes and jobs may be shared by different computing facilities that are separately administered creates a problem of security. How will systems administrators authenticate and set permissions for tasks that do not originate from within their administrative control? How is trust established between different administrative domains? Different administrative domains also mean there is no guarantee of the immediate availability of resources, since one administrator has no control over the equipment administered by another party. Furthermore, if there is a hardware or software failure at a different administrative domain, there is nothing the local administrator or grid software can do to correct it. Unlike centrally administered cluster software, a designer of grid software must take all these issues into consideration.

3. The issue of heterogeneous architectures is the most problematic when designing grid software. Usually, programs compiled for a specific architecture cannot be run under normal circumstances on a different architecture. Even if the architectures are identical, it is seldom possible to run programs which are compiled for different families of operating systems such as Microsoft Windows and GNU/Linux. To get around this, users of heterogeneous systems standardise on a single portable bytecode-based programming language such as Java, Perl or Python.

These three points are an indication of a need for different approach to distributed computing when thinking about grids. The software and algorithms used for clusters cannot be totally reapplied without consideration for the preceding issues.

Most grid computing middleware is implemented as userspace daemons, libraries and programs. For example, Globus [21] is a collection of services that comprises a resource manager called GRAM (Globus Resource Allocation Manager), a communication library called Nexus, a directory service for state information called MDS (Metacomputing Directory Service), a remote data access service called GASS (Global Access To Secondary Storage), a monitoring tool called HBM (Heartbeat Monitor), a security and authentication framework called GSI (Globus Security Infrastructure) and an executable programs management service called GEM (Globus Executable Management).

Of each of these services, almost all of them introduce a set of APIs for the programmer to use when designing grid programs. The disadvantage of this approach is the introduction of complexity and cruft, especially for the user who needs to create new programs designed specifically for the grid, as well as users who wish to run their existing applications on the new grid environment. Furthermore, grid software currently available runs on top of existing operating systems that are completely unaware of the existence of grid users and grid processes that are executing on top of it. Therefore, the process management of the OS cannot schedule or handle these grid tasks in a way that would benefit the grid application. The emergence of these grid applications has created unique new requirements for process management, which most mainstream operating systems have no support for.

2.3 Operating Systems Support For Grid Computing

To argue the case for specific support for grid computing in operating systems, we shall briefly consider the different kinds of traditional operating systems and their different levels of support for networking and distributed resource sharing. Then we will identify another subclass of operating system which complements existing types of operating systems; the grid operating system.

2.3.1 The Centralised/Local Operating System

The centralised/local operating system represent a class of operating systems without network support and were common in the earlier days of computers before networking became popular. They can be single user or multiuser, but lack a network stack to communicate with other computers. The early versions of Unix were entirely centralised and local operating systems. A local operating system generally implements the four basic components of operating systems : file management, device management, memory management and process management.

2.3.2 The Network Operating System

The network operating system is an OS which implements a network stack, and implements several network services such as remote file or print servers. All modern operating systems such as Windows NT and Unix have networking functionality, and thus qualify as network operating systems. Unix and Unix-like operating systems like GNU/Linux usually come with many networking protocols, but the most popular of protocols in the Internet era is TCP/IP. Unix-like systems implement TCP sockets as an extension of the file and device management components; every network connection socket can be treated as a file to which we can read and write from.

The network operating system grew out of local operating systems which were given networking support. Therefore, they are generally not designed for running on a collection of computing elements which act as a cohesive whole. A distributed operating system is an OS for multiprocessing and multicomputing environments and is run as a single system image.Tanenbaum and van Renesse [75] outline several fundamental characteristics of the network operating system as opposed to the distributed operating system:

1. Each computer is running its own private operating system.
2. Each user logs in on each computer individually without a single sign-on nor a single system image which dynamically allocates CPU usage to the user from a pool of available CPU's.
3. File placement on different computers need to be managed manually; copying files from one machine to another requires manual network copy.
4. Very little fault tolerance; if some computers are out of commission, the users on that computer cannot continue working.

2.3.3 The Distributed Operating System

The key distinguishing characteristic of a distributed operating system from the network operating system is the transparency of its operation on multiple computers. The user should be able to see the distributed operating system as a single system image, where every computing resource is represented as part of a whole. The user should authenticate and log in only once, be able to access files on a local or remote machine anywhere in the system, run a process on any CPU, and the failure of a single component should not cripple the system. All resources, whether files or CPUs, must be able to be accessed with the same usage semantics regardless of the physical machine they reside on.

There have been several experimental distributed operating systems, such as Amoeba [73], Sprite [57] and Plan 9 [60]. Sadly, they have not gained widespread acceptance beyond the OS research community. Even so, modern network operating systems have come a long way since early network operating systems were introduced, and many of the distributed operating systems' features have been incorporated into them. Modern Unix-like systems can be equipped with NIS [34] or LDAP-based ([84], [80]) single system sign-on and authentication, and various distributed filesystems have been introduced such as NFS [65, 59] and Coda [66, 38]. Symmetric multiprocessor (SMP) and NUMA machines also incorporate dynamic CPU allocation across multiple CPUs. The Linux operating system supports all these technologies, and therefore is used on clustering and parallel processing platforms. However, the support for automatic distributed process management and CPU allocation outside of proprietary NUMA machines or SMP machines remains missing in mainstream GNU/Linux distributions.

2.3.4 The Grid Operating System

The current definitions of "local operating systems" and "network operating systems" are inadequate to describe an operating system with grid support. It might be convenient to group grid-enabled operating systems together in the "distributed operating system" category, but there are several aspects of grid computing which are not addressed by the definition of a distributed operating system.

A distributed operating system implies a single administrative domain (where a single party is responsible for controlling access, maintaining and granting permissions to users), whereas a grid can encompass many different administrative domains that want to pool their selected resources together. This has two consequences:

1. A grid can be a very decentralised entity with different authentication systems, different administrators and different geographical locations. A decentralised system defined by rules on the sharing of distributed computing resources is referred to as a *virtual organisation* [23]. Thus, though there is a mutual agreement of the sharing of a resource pool, not all resources belonging to each party are shared, and perhaps not all the time as well.

2. A distributed operating system assumes that all resources on the system can be allocated and scheduled with full authority. This is not the case in a grid resource pool which may encompass different administrative domains have their own allocation schemes and access control mechanisms which can not be overridden by another administrative domain. Hence, there is no central authority in grid systems.

According to Mirtchovski et. al [53], current operating systems such as Windows and the Unix variants were designed predating the advent of networking and the internet. Therefore, they are poorly suited for grid computing. Thus, a need for a grid operating system is a real one. Just as traditional operating systems simplified the usage of complicated assorted hardware resources by creating abstractions for them which a user can use transparently and easily, it is hoped grid operating systems will do the same for the eclectic mix of distributed resources on the grid.

2.4 Resource Discovery

The first step in the utilisation of the grid is resource discovery. In a fully decentralised distributed network, the task of querying a global state is done by first determining the existence of other nodes in the network. The process of each node on the network discovering other nodes that want to cooperate with it for distributed processing is the solution

to what is known as the resource discovery problem. The system can be represented as a directed graph, where each vertex represents a node. A directed edge from node A to node B represents that node A knows about node B and B is said to be A's neighbour.

The resource discovery problem was described by Harchol-Balter *et al.* [29] in 1999. The model for resource discovery proposed had the nodes in the distributed system partaking in a series scheduling events at synchronous intervals of unspecified length called "rounds". In each round, each node would send a list of its neighbours in to other nodes, and a node receiving a list of neighbours would be able to create connections to new nodes previously unknown to it and add the new nodes as neighbours. After a certain number of rounds, the graph would be fully connected (all nodes know about every other node).

It is important that a robust resource discovery algorithm is used to give each node a picture of the global state of a distributed system. This information must be kept as current and as accurate as possible for each node. However, this must not be done at the expense of flooding the network, sending too many messages, or taking too long to complete. Harchol-Balter *et al.* outlined three metrics for evaluating the performance of resource discovery algorithms:

1. The number of rounds taken for the graph to reach full connectivity.
2. Pointer communication complexity – the number of "pointers" that is communicated during the course of the algorithm.
3. Connection communication complexity – the number of connections that are made during the course of the algorithm.

The Name Dropper algorithm was proposed by Harchol-Balter *et al.* in the same

paper. It works as follows; consider a node v, where $\Gamma(v)$ is a set of all nodes which v knows about (the set of neighbours). In each round, each node v transmits $\Gamma(v)$ to *one* randomly chosen node u where $u \subset \Gamma(v)$. Upon receiving $\Gamma(v)$, node u will update its own neighbour list $\Gamma(u)$ with new information from v, i.e. $\Gamma(u) \leftarrow \Gamma(u) \cup \Gamma(v)$. The graph will achieve complete connectivity very quickly in $O(log^2 n))$ rounds with high probability, whereas pointer communication complexity is $O(n^2 log^2 n)$ and connection communication complexity is $O(nlog^2 n)$.

Name Dropper is similar to gossiping [30] algorithms which is used to broadcast information to a set of nodes. However, unlike gossiping, Name Dropper does not require each node to know about every other node in advance, nor does it require a fixed communications network.

Zinc uses Name Dropper to propagate resource discovery and state information updates across administrative domains. Due to its efficiency and simple implementation, Name Dropper performs very efficiently for fast information propagation.

2.5 Grid Scheduling Stages

According to Schopf, the scheduling of a job will go through the following stages in the grid[67]:

1. Resource discovery
    - Authorisation filtering – Restricting the search for resources that are only authorised to be used by the user.
    - Application definition – Defining the requirements of the user's job to select the appropriate resources

- Minimum requirements filtering – Eliminating the resources that do not meet the minimum requirements criteria from the set of resources to choose from.

2. System selection

   - Information gathering – Collecting state information from the grid, which is mainly derived from a Grid Information Service (GIS) and the local scheduler.
   - System selection – Deciding which system matches up with the requirements for the application. Examples of approaches for this are Condor Classads[62], multi-criteria [42, 41] and meta-heuristics [50].

3. Job execution

   - Advanced reservation – An optional step, users may opt to reserve resources in advance.
   - Job Submission – The user submits the job to the system. Currently there is no standardised way of doing this in the different grid middleware implementations.
   - Preparation tasks – Ensuring the files needed are in place, claiming a reservation, or whatever "preparation" steps needed to run the job.
   - Monitoring progress– Enabling the user to track the status of his or her job
   - Job completion – When the job is completed, the user is notified.
   - Clean-up tasks – Removing temporary files, retrieving data files, resetting configurations or other miscellaneous "clean-up" procedures.

Note that though Schopf calls the first stage "resource discovery", the sub-stages listed are have more to do with user authentication and user requirements collection than the actual "discovery" of distributed resources as described by Harchol-Balter in [29], which is categorised by Schopf into a subset of "system selection".

With the Zinc program we have not attempted to implement the full functionality as described above, but instead a subset of the essentials to provide a prototype to make grid scheduling work on our extended Linux kernel. In particular, users do not need to specify their requirements, thus there is no explicit requirements matching done. This is to keep our prototype simple enough to test the effectiveness of process migration and the Zinctask system selection algorithm.

2.6 Distributed Scheduling Algorithms

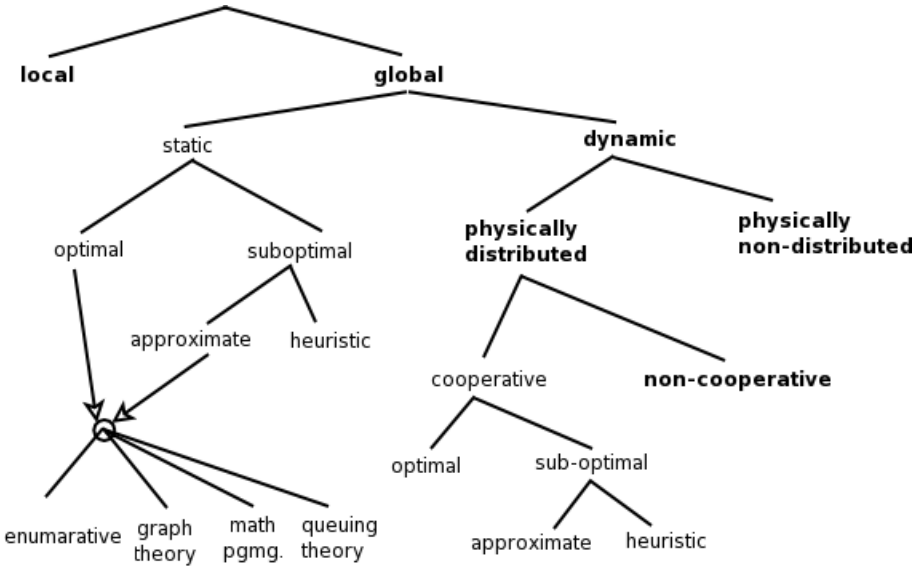

Figure 2.1: Casavant and Kuhl's Taxonomy of Task Scheduling

Casavant and Kuhl [9] devised a generalised taxonomy for the classification of task scheduling in distributed systems (reproduced in figure 2.1), which we have applied to our investigation of a suitable scheduling policy and mechanism for a grid operating system. Casavant and Kuhl distinctly define local and global schedulers as two separate logical entities in their taxonomy. We have highlighted the characteristics of the Zinc scheduler in bold, and is discussed in the following subsections.

2.6.1 Local and Global Scheduling

Local scheduling is the operating system scheduler responsible for the timesharing of the CPU (or CPUs) within a single computational node, whereas global scheduling refers to the placement of jobs or processes on the distributed system. A grid operating system addresses issues in both type of schedulers, as the OS kernel should be aware of "grid processes" (i.e. processes that are started by grid users and may migrate to other nodes) and how to manage them even at the fine-grained CPU scheduler level. The placement of jobs on distributed nodes can be handled in userspace, but the userspace scheduler should interact with the local CPU scheduler to handle grid scheduling across the system. This two-level hierarchy may be viewed as a distributed variant of two-level scheduling as described by Tanenbaum and Woodhull in [77], with a difference; instead of moving processes around from disk and main memory, the top-level scheduler migrates them to other nodes. In the Zinc-enabled version of GNU/Linux, the kernel scheduler uses a high-throughput process scheduling algorithm [32] and process migration is controlled by the top-level distributed userspace scheduler. The top level userspace scheduler interacts with the CPU scheduler by setting priority values for grid processes in the kernel to reflect administrative domain policies.

2.6.2 Static Vs. Dynamic

Static scheduling is the formulation of a scheduling strategy for the system with information regarding all the processes that are going to be run in the system as well as the system state being known *a priori*, or before the execution even begins. Dynamic scheduling does not assume any such information is available, and instead processes are managed as they begin executing, which allows us to have a more realistic assumption about the information we have when we start executing programs. Local schedulers for modern operating systems

such as GNU/Linux use dynamic scheduling, and thus our model of the grid operating system also uses a dynamic local scheduler. Thus, we adopt a dynamic scheduling strategy for our grid operating system at both the local and global levels.

2.6.3 Centralised Vs. Distributed

A centralised or non-distributed scheduling mechanism refers to a central authority which ultimately decides scheduling decisions for the distributed system, whereas a distributed one does not rely on single authority but is decided collectively by a number of different decentralised scheduling authorities. For Zinc, we have adopted both strategies; for a single administrative domain, the strategy is centralised, but between administrative domains distributed scheduling takes place. This reflects the nature of grid administrative domain authority; within an administrative domain, a centralised policy can be enforced for all nodes, but between administrative domains on the grid, there exists no central authority. Dandamudi and Lo [12] found the performance of this hierarchical scheduling scheme inherits the advantages of both centralised and distributed systems while minimising their drawbacks.

2.6.4 Cooperative Vs. Non-cooperative

For distributed dynamic global scheduling policies, the strategy can be either cooperative (every scheduling entity works for a single goal) or non-cooperative (each scheduling entity makes decisions for its own resources with full autonomy). For scheduling between administrative domains in Zinc, we chose the non-cooperative model, as it too reflects the nature of the grid of limited, controlled sharing of resources. Therefore, there is no explicit "global goal" of grid schedulers, only the fact that they may agree to work together by making requests for resources across each domain. The requests are only granted if it is

convenient for the administrative domain owner (i.e., it's not too overloaded).

2.6.5 Source-initiative Vs. Server-initiative

This classification scheme is not explicit shown in Casavant and Kuhls' taxonomy, but is suggested as a narrower, specific classification of load-sharing. The distinction between source-initiative and server-initiative algorithms was proposed by Wang and Morris in [82]. This classification determines who takes the initiative in load-sharing. A source-initiative strategy is when the source node searches for a suitable host to send a job. The server-initiative however, actively "looks for work" by selecting jobs from different sources which it is willing to process. Wang and Morris conclude that load sharing algorithms are an important design decision, for they found wildly varying performance with different strategies. Also, server-initiative algorithms potentially outperforms the source-initiative one when given the equivalent amount of state information. However, performance (measured by Wang and Morris with a metric called "Q-factor" which is a function of mean response time) is sensitive to the amount of communications overhead, the number of servers and the number of sources. Furthermore, while Wang and Morris' observations hold true for non-preemptive scheduling algorithms, Dandamudi [11] found the source-initiative strategy to provide better performance for preemptive round-robin algorithms. Other researchers have also found that source-initiative algorithms potentially perform better than server-initiated ones [70, 87]. For Zinc, one of our design goals is to minimise the amount of migration that needs to take place. We also use preemptive round-robin scheduling instead of a FCFS queue. Therefore, processes will not be migrated if there is no need for that to occur. Hence, we chose the source-initiative algorithm.

2.7 Process Checkpointing And Migration

2.7.1 Checkpointing Techniques

There have been several implementations of process checkpointing mechanisms, and they can be classified into either kernel-level checkpointing or user-level checkpointing schemes. User-level checkpointing implementations usually requite that the application be built with checkpointing in mind. This is done in 3 ways:

1. Compiler-assisted – The user manually specifies points in the source code where checkpointing can take place, and the compiler produces a binary which is checkpointable in userspace.

2. Checkpointing library – The user program is linked to a special userspace library which enables checkpointing, thus does not require a special compiler.

3. Runtime-assisted – The programming language run-time system (for languages that have one) transparently supports checkpointing.

Kernel implementations of checkpointing and migration mechanisms usually provide a more transparent facility for users. This is because a lot of process state is stored in kernel data structures, and thus having the cooperation of the kernel simplifies dumping and restoring this state information. Kernel implementations of checkpointing is usually a straightforward dump of the contents of the process state and virtual memory map into a file. Restoring a checkpoint involves reading the dump file and setting up the virtual memory areas again, while resetting the registers and kernel data structures. Unlike userspace checkpointing which requires tricks to make the kernel start a program half-way, restoring a program in kernelspace is easier to do since we can manipulate kernel data structures directly.

2.7.2 Process Migration Overview

Process migration is "the act of transferring a process (in execution) between two machines" [51]. Checkpointing is defined as "a technique for saving process state during normal execution and restoring the saved state after a failure to reduce the amount of lost work" [81]. Checkpointing and migration often go together as checkpointing can be used to help implement the process migration mechanism.

Checkpointing and migration have traditionally been used to implement load balancing in a distributed system. As computational grids become more common, the need for a transparent checkpointing and migration facility arises as jobs submitted to the grid need to be organised and possibly redistributed across the computational grid for maximum efficiency and throughput. This calls for processes to be migrated in the middle of execution, or preemptive migration.

2.7.3 Motivation For Process Migration

An excellent survey on process migration has been done by Milojičič *et al.* [51] in which the subject of process migration was discussed in detail, along with a discussion on many different process migration systems already implemented. Milojičič outlines six major goals of process migration:

1. Accessing more processing power – mainly used for load distribution and load balancing.

2. Exploitation of resource locality – moves processes closer to the resources they need for faster access.

3. Resource sharing –moves processes that need special hardware or software to the

place that has it.

4. Fault resilience – processes can be migrated from a failing node to another node, and checkpointing can also act as a failsafe mechanism in the case of sudden shutdown.

5. System administration – system administrators can move important processes from a machine that needs maintenance.

6. Mobile computing – users may want to migrate a process from a remote location to their computer at another location and migrate it back when they're done.

All these goals are also desirable in a grid computing environment which would also benefit from process migration and checkpointing in the management of distributed processes and jobs. It would be very helpful if a long running job could be automatically checkpointed for recovery in case of failure. It is also efficient to migrate compute intensive jobs to more powerful computers as they become available. Grid computing usually involves the execution of long-running computational processes. Applications such as molecular modelling, DNA and protein analysis, mathematical algorithms and other scientific applications which are the target usage of grid technologies usually run for hours or days, even on modern hardware. Therefore, it would be advantageous if the execution state of these applications could be captured and saved to disk, either for backup purposes (so that work won't be lost if there is node failure) or for process migration. Furthermore, in a grid environment, availability of resources is not always guaranteed, so the grid scheduling system needs a some way of performing preemptive process management in an always-changing computing environment for the sake of efficiency and maximising throughput.

Despite the benefits, there are also drawbacks to process migration:

1. Perhaps the biggest drawback is cost of migrating large process images can be high.

A checkpoint image with a virtual memory size of 1 gigabyte or more will be very costly to migrate. However, the continual improvement of network hardware which currently cheaply provide gigabit speeds may improve the situation. If the migration times can be brought to reasonable time, it may still beneficial to migrate long-running jobs that take magnitudes longer than the transfer of its checkpoint image, for reasons of load balancing and maximising throughput. Zinc addresses this problem by preferring processes with smaller virtual memory sizes when choosing processes to migrate, and automatically disregarding remote hosts with insufficient bandwidth.

2. Process migration is also difficult to implement correctly while using all the features of modern operating systems such as sockets, pipes and shared memory. Single administrative domain migration systems such as Condor and MOSIX provide inter-process communication support via home nodes that support the execution of remote processes. However, for wide-area grid migration, home nodes which require a persistent network connection are troublesome to maintain over the internet where high lag times may impede performance of the process. Thus, we have assumed processes that are going to be run do not make use of inter-process communication, and many numerical and scientific programs which the grid will be primarily used do not need IPC or other advanced Unix facilities (Milojičič *et al.* [52] support this view). However, it is a weakness which needs to be addressed in the future.

Eager *et al.* [16] argue that there are no conditions where process migration would offer substantial performance improvements over non-preemptive strategies, and recommend non-preemptive migration as a better strategy for load distribution, citing lower costs. However, Harchol-Balter and Downey argued against this conclusion in [28], stating that the theoretical model used by Eager *et al.* is flawed by taking into account processes with

zero lifetimes (which do not exist in the real world), do not recognise the costs involved in initial placement of processes (non-preemptive migration), and assumes an exponential process lifetime distribution, which does not reflect the process lifetime distributions of real systems. Harchol-Balter and Downey argue the following in favour of preemptive migration:

1. Migrating long jobs away from busy hosts helps both the long job and other short jobs that share the host, and a busy host is expected to get many short jobs.

2. Preemptive migration outperforms non-preemptive migration even when memory transfer costs are high, because non-preemptive strategies need to predict which processes will be long-lived. The cost of an incorrect prediction will cause other processes on the node to slow down due to the long-running CPU hog that was inappropriately assigned to the node. A preemptive migration strategy can correct placement mistakes with migration.

3. Despite preemptive migration being difficult to implement, some systems implement it for reasons other than load balancing.

2.7.4 Migration Algorithms

Process migration is dependent on the ability to suspend the operation of a running process and transfer its state to a remote machine and resume execution. According to some researchers such as Eskicioğlu [17] and Zayas [85] , the size of the virtual memory or the used process address space is the main determinant of the performance of the distributed system. There are several methods of implementing virtual memory transfer, four of which are outlined by Eskicioğlu:

1. Entire virtual memory transfer – The most straightforward scheme where the entire process space is transferred to the destination machine. The process is suspended on the source machine before the virtual memory image copying takes place, and only when all the pages are transferred to the destination host, the process is resumed.

2. Pre-copying – The memory pages are evicted to the target machine while the process is still executing on the source, and dirty pages are copied over as needed until a threshold point when the number of dirty pages is at a minimum. Then the process is suspended and the remainder of the pages are transferred, after which the process resumes on the destination host.

3. Lazy copying/copy-on-reference – The process is suspended on the source and all state information except the virtual memory is copied over to the target machine. The process is resumed on the destination host and the pages are copied on demand when each of them are accessed.

4. Enhanced copy-on-reference/flush – Almost the same method as regular copy-on-reference, the only difference is that dirty pages from the address space is copied to a file server after suspension before any part of the process is transferred to the target machine. Then the remaining state information is sent to the destination host where it is restarted, and the missing pages are copied on reference from the file server.

Richmond and Hitchens [64] proposed another process migration algorithm similar to the ones above called *migration by post copy*, where the process is suspended on the source host, the minimum necessary state information is transferred to the destination host like the lazy copying algorithm. However, the source host also simultaneously transfers the rest of the virtual memory pages as the process starts to execute. If there any specific pages which the execution process needs, they are requested by the destination node and

the source node also sends the requested pages as needed.

By it's very nature, process migration works only for homogeneous architectures. Arbitrarily saving the process state of a program on one architecture and translating it into another is a difficult problem, but there have been efforts to attempt to solve it. Usually such solutions involve virtual machines which provide an identical environment on which programs can run on, regardless of the host architecture. This is the approach taken with Java process migration and mobile agents. Another approach is modifying the source code of a program to accommodate checkpointing, and since checkpointing semantics is supported at the source code abstraction and compiler level, different architecture-specific binaries can be generated from it. Such was the approach taken by Theimer and Hayes [79], Strumpen and Ramkumar [71] and Ferrari *et al.* [18].

2.8 Systems Which Implement Process Migration

The following are examples of systems using process migration implemented in slightly different ways. While by no means is it an exhaustive survey, the following list highlights various implementation techniques for process migration.

2.8.1 Condor : A Userspace Implementation

Condor[46] was developed at the University of Wisconsin-Madison to maximise the utilisation of idle workstations on their campus, thus delivering a source of computational cycles that was otherwise untapped. It's the idea of efficient utilisation which led to high-throughput computing [48, 49], which has the goal of sharing resources effectively on the grid and doing as much work as possible with those shared resources. The purpose of process migration in Condor is to provide workstation autonomy; a user who starts to use his

or her own workstation would want the CPU to himself or herself. Thus, guest processes running on the machine need to be evicted and run elsewhere. Process migration in Condor [47] is implemented in userspace. A custom C library with specialised system call wrappers handles the process' interface with the operating system, hence allowing for process stubs to be created and communicate with a remotely migrated process. In the actual creation of the checkpoint, process state is extracted from a core image of a process, which the kernel can be instructed to dump to a file. From the core image, a custom `a.out` file is constructed by Condor which contains special instructions (specifically, a signal handler) to trick the kernel into restoring a process' state.

2.8.2 Amoeba : A Microkernel Implementation

Amoeba[55, 73, 76] is a distributed operating system developed at Vrije Universiteit in Amsterdam since 1981. Amoeba employs the concept of "processor pools", where computers with high processing capabilities provide CPU time to users, who submit jobs from terminals. Similarly, there are file and directory servers which provide storage and other facilities. It's radical design meant Amoeba was only partially compatible with existing operating systems (i.e., Unix) and software had to be ported to it with some effort.Initially, Amoeba did not have process migration, and it was added by retrofitting the Amoeba kernel with the capability to suspend and transfer address space [70]. It is a microkernel-based operating system, where the kernel maintains a very minimum of state information on running processes, which simplifies the design of the process migration algorithm as not a lot of state needs to be transferred. Also, Amoeba is designed from the ground up to be a distributed operating system. All inter-process communication takes place via remote procedure calls, which themselves are implemented on top of a location-independent networking protocol called FLIP [35]. This location-independence makes the transfer of processes easy across nodes in an Amoeba system, as there is no need for explicit rerouting

of messages from a home node.

2.8.3 Sprite : A Monolithic Kernel Implementation

Like Amoeba, Sprite[57] was also designed from the ground up as a distributed operating system. However, unlike Amoeba, Sprite tries as much as possible to resemble a traditional Unix-like operating system to the user. Also, the Sprite kernel is of traditional monolithic design, which is argued to have a simpler implementation and higher performance compared to microkernels. Sprite incorporates process migration as a fundamental service which the operating system provides[13]. Sprite uses a number of techniques to ensure transparent migration:

1. Virtual memory images of processes are transferred via the enhanced copy-on-reference algorithm.

2. Uses a combination of system call forwarding and kernel state transfer to ensure the execution environment remains the same for the migrated process

For a comparison of Sprite and Amoeba, see [14].

2.8.4 MOSIX : Extending An Existing Unix-like Kernel

MOSIX [3] is a project that implements process migration on BSD and Linux, by adding kernel modifications. MOSIX was initially intended to work on clusters, where a persistent network connection is available to all nodes. MOSIX processes which migrate maintain communication with a stub or "deputy" on the home node, and message forwarding is done to between the home node and the remote node to maintain transparency. Like Amoeba, MOSIX uses a "central pool of processors" model that assumes the nodes used

for computationally-intensive processing are dedicated to the job, and thus does not evict processes for workstation autonomy. MOSIX continually tries and balance the load across nodes in the system when it is running. MOSIX has recently been expanded to work on clusters of clusters which resemble grids.

2.8.5 Xen : Virtual Machine Migration

The Xen [15] virtual machine (VM) monitor is an open source paravirtualisation tool which allows virtualised instances of entire operating systems to run concurrently on the same machine. The Xen virtual machine monitor allows the migration [10] of virtual machines across physical machines, which effectively migrates the entire operating system together with all the processes running on it. To evict the virtual memory pages of the running VM, the pre-copy migration algorithm is used by Xen to migrate most of the pages and transfer any dirty pages left over as they are accessed. When the migrated VM is restarted, the VM at the original source is suspended and any leftover pages are copied over on reference.

This technique of migrating entire operating system instances has the benefit of eliminating nearly all residual dependencies between source and target machines. However, the problem of IP address reassignment and large disk image transfer makes this method unsuitable for wide area migration. Also, migrating all processes in a given operating system instance is too coarse-grained for grid computing load distribution.

2.9 Grid Operating Systems

The concept of a grid operating system has not been explored much in literature. We are only beginning to see facilities for grid computing integrated or retrofitted into existing operating systems such as the MOSIX extensions for Linux and the Plan 9 distributed

operating system from Bell Labs.

2.9.1 Federated MOSIX Clusters

In 2005, the MOSIX research team published a paper on the use of MOSIX to construct a grid of "federated MOSIX clusters"[4], which allows the use of MOSIX to migrate processes within a grid. In this paper, grid-related problems such as resource discovery, precedence (priority) order of guest and local processes and a virtual run-time environment was discussed. MOSIX now supports intra-cluster migration, but for a inter-campus grid with a high-bandwidth network.

2.9.2 The Plan 9 Approach

Plan 9 from Bell Labs is a distributed operating system developed in the late 1980's at Bell Labs, from the same development team that designed the original Unix. Plan 9 takes the concepts of Unix and extends it several steps further, such as the "everything-is-a-file" philosophy in Unix, which in Plan 9 applies to almost all resources in the system. These resources, whether they are local or remote, can be accessed uniformly through the filesystem namespace. Plan 9 was designed following the "resource pool" model which is similar to Amoeba and MOSIX, though there is no support for load sharing or process migration at the time of writing.

Mirthovski *et al.* [53] argue that the reason Plan 9 is an ideal operating system for the construction of computational grids, for the basic mechanisms for resource sharing is fundamentally built into the operating system with clean, simplified abstractions for accessing the resources:

1. Distributed authentication is possible with authentication agents and servers.

2. Private namespaces for processes maintain a virtualised environment which maintains security.

3. Resource discovery is done simply by sharing namespaces where a view of remote resources can be mounted as part of the filesystem, thus allowing them to be inspected using standard Unix tools such as `cat` and `grep`.

4. There is no need for explicit data management utilities, since the sharing of namespaces allow the transfer of data across the distributed system which is done automatically by the operating system.

It remains to be seen whether Plan 9 will gain greater acceptance by the grid computing community. While it's innovative use of namespaces provide an interesting mechanisms for resource discovery and data management, it still lacks mechanisms to control and distribute system load.

With Zinc, we address the issue of providing distributed process management for GNU/Linux, and in the future add the concepts of namespaces as pioneered by Plan 9 to make the system more complete.

2.10 Summary

The field of distributed computing has given birth to many ideas which can be applied (and adapted) to the emerging new paradigm of grid computing and grid operating systems. Resource discovery, distributed scheduling and process migration all come into play when designing a process management facility for a grid operating system. In this thesis, we will attempt to answer the question as to whether or not process migration is worth the cost of implementing with respect to the goals of grid computing, one of which is high-throughput

computing. Based on the work of established by researchers in the field of distributed computing, we have designed our algorithms to be efficient and robust. The performance of these algorithms have been tested experimentally, and are presented in chapter 6.

CHAPTER 3

DESIGN GOALS

3.1 Introduction

This chapter encompasses the design goals of the creation of Zinc, a software extension of Linux to make it more "grid aware". Section 3.2 presents the overview of what the Zinc framework tries to accomplish. Section 3.3 takes a look at the design goals of Zinc, and section 3.4 presents details on the handling of wide area migration issues. Section 3.5 takes a look at the performance metrics used to evaluate our system, followed by a conclusion for the chapter in section 3.6.

3.2 Design Overview

3.2.1 Extending GNU/Linux To Become A Grid Operating System

The Linux kernel is a monolithic kernel by design, and to extend the functionality of the operating system, one needs to be careful as side effects could break other subsystems. Linux follows the general design of any modern operating system, comprising the following:

1. Filesystem management
2. Device and I/O management
3. Memory management
4. Process management

There are several other ways to break down the functionality of the Linux kernel, but for the purpose of this thesis, we shall use the above generic taxonomy described by Nutt [56] to illustrate the different responsibilities of any modern operating system kernel. GNU/Linux is a network operating system, with its userspace programs offering a myriad of network services such web servers, ftp servers, mail servers and the majority of all the other network services that are available to today. The kernel however remains largely a centralised entity, although the most recent Linux kernel releases have forayed into distributed system territory. The process, memory and device managers have supported symmetric multiprocessing since version 2.0, and the filesystem manager supports various distributed filesystems such as NFS [65, 59] and Coda [66, 38] by default. The newest versions of the kernel at the time of writing, the 2.6 series, has added support for NUMA architectures.

This taxonomy is also used by Goscinski [25] to illustrate how a distributed operating system can be implemented by placing interprocess communications as a layer which services any or all of the components of the distributed operating system. Hence, we can have distributed filesystems, distributed devices, distributed memory and distributed process management. To what degree an operating system can be considered "distributed" depends on how many of these basic components are supported in the distributed paradigm, and how much integration is done between the components. A fully distributed operating system will appear as a single-system image when running on multiple computing nodes, where processes and users have transparent access to all resources on the system. However, a GNU/Linux operating system only partially implements distributed functionality.

It is useful to consider the features of traditional distributing operating systems when pondering the design of a grid operating system. One of the primary defining characteristic of a grid is the ability to selectively share resources over a distributed network of computers. The Zinc framework will only be implementing grid process management, as outlined by

the scope of the research in Chapter 1.

3.3 Goals

In the design of Zinc, the following factors were taken into consideration that support the idea of a grid operating system:

1. Transparency
2. Improved throughput of jobs in the entire system
3. Adaptability to a varying resource pool
4. Minimal residual dependencies
5. Decentralisation

These points provide the motivation for the design decisions that were made for Zinc. Each of these points are expanded on and explained in the following subsections.

3.3.1 Transparency

The goal of any operating system is to provide a layer of virtualisation on top of the hardware, as to shield the user from having to deal with low-level details of the implementation of services. Therefore, a grid operating system should provide transparent facilities for all process execution activities that are related to its participation in the grid.

According to Tanenbaum and van Steen [74], in any distributed system there are several concepts of transparency:

1. Access transparency – provides an abstraction for data structure representation

2. Location transparency – the location of a resource on a distribution system is abstracted into a unified view for the user

3. Migration transparency – a resource may be moved across physical locations on the distributed system and can still work the same way

4. Relocation transparency – like migration transparency, but with the added requirement of being able to move a resource while it is in use

5. Replication transparency – multiple copies of the resource exist, but appears to the user as a single copy

6. Concurrency transparency – access to a resource can be done by concurrently by different users without the users being aware of the sharing

7. Failure transparency – the user need not know or does not notice that a resource has failed and has restored itself and corrected the failure

8. Persistence transparency – the user does not know whether a resource lies in primary or secondary memory

Tanenbaum and van Steen also state that there are degrees of transparency on a distributed system. For Zinc, we have tried to provide a high degree of migration and relocation transparency of processes and the files the process works with. In particular, the following mechanisms are done without user intervention:

1. Transparent process checkpointing

2. Transparent host selection

3. Transparent process migration

All of these are inter-related, before a process can be migrated, the process needs to be halted and the state needs to be saved into a checkpoint image. Then, another host must be found, and the checkpoint image needs to be sent to the target host for restarting.

The primary goal of a kernel-level process checkpoint and restore mechanism is to provide near-total transparency of checkpointing. Although checkpointing is quite a well-researched and widely available facility, most grid middleware do not come with checkpointing mechanisms for the enabling of process migration. A notable exception to this is Condor[47],which provides application level-checkpointing by relinking target applications with a custom C library. The C library would then make custom function calls that redirect all input/output via stubs if the application running gets migrated away from its home node. Although this is enough for most purposes, it makes a number of assumptions about the application that is to be run:

1. It is dynamically linked
2. It uses a specific version of a C library
3. A persistent network connection is available between computing nodes

In the design of a general purpose computing environment, we feel it is not desirable to restrict the developer of applications to a particular version of a function library, or to introduce new APIs to the programmer. Therefore, checkpointing is best handled at the kernel level, which does not require any specific linking provisions for the applications in userspace to take advantage of the checkpointing capability. This is because all the checkpointing logic can be encapsulated inside the kernel, with system calls being the only method of invoking the checkpoint mechanism. This frees the programmer of having to use specific libraries or specifying checkpoint points in the source code. This is the level of

abstraction that an operating system should provide for the user.

Once a program has been checkpointed, it can then be transported to another host for restart. The process of host selection and the copying of the checkpoint image are done in userspace, via the Zinc userspace scheduler. Therefore, the policy for process migration can be set in userspace without touching kernel behaviour.

3.3.2 Improved Throughput

One of the goals of grid computing is to pool together distributed resources and make the most efficient use of this shared pool. This has led to the development of complex scheduling and resource allocation mechanisms to handle job management on the grid. There have been resource allocation policies based on expected running time [43], deadline-based scheduling ([8], [72]), expected resource requirements of jobs [37] and also economy-based resource management [7]. These different strategies have different emphasis when dealing with the problem of resource allocation. Policies based on deadlines and running times are mainly concerned with the time-constraint needs of the user, while economy-based strategies take into account the monetary costs of using shared resources and try to optimise the scheduling for cost-effectiveness. However, there is an emerging paradigm of resource allocation that has grown in unison with grid technology, which is called high-throughput computing or HTC. HTC-centric scheduling policies tend to try to optimise the utilisation of pooled resources for all users, rather than to try and set hard-and-fast deadlines on the time of completion of jobs. According to Livny and Raman[49], the characteristics of HTC is that of opportunistic scheduling, where resources can be available at any time without advance notice, and the system must be robust enough to schedule jobs to take advantage of dynamic resource availability. The objective of this is to get as much work done as possible in the grid environment, while honouring the priority of the owners of

the resources. Therefore, the design of Zinc follows this paradigm of resource allocation though it de-emphasises the importance of workstation autonomy which is one of the design goals of Condor. This is because we wish to create a distributed computing model of dedicated networked grid computing machines and not merely "scavenge" for free CPU cycles from desktop users. This model is based on the assumption that resource owners are participating in the Zinc-powered grid with the specific intent of building a shared computing infrastructure, and not relying on spare compute cycles from random users. This design goal borrows from the design philosophy of Amoeba which has a dedicated processor pool for sharing amongst users [14].

It is interesting to note that the emphasis on throughput does not make the scheduling policy mutually exclusive with other scheduling policies. Since the notion of priority is ever present in a HTC environment, it is possible to use priority as a weight when designing scheduling functions. This weight can be used as a representation of urgency (a higher priority can be assigned to a user who requires tasks to be completed soon) and also cost (a higher priority can be assigned to users who are willing to pay more for better service). However, Zinc was not designed to guarantee that jobs will complete by a certain deadline, since the problem of accurately ascertaining the running time of a particular program is difficult if not impossible.

3.3.3 Adaptability To Dynamic Resources

A grid computing environment encompasses a scale which is unlike ordinary computing clusters or massively parallel processing machines (MPP's). Clusters and MPP's are tightly-coupled, administered by a single entity, and the set of available resources is controlled and of a constant number. Grids are more loosely-coupled, and perhaps spans many administrative domains, and the number and quality of resources that are available vary

with time. A scheduling policy that works well on a multiprocessor or cluster system will not work well at all on a grid. This is because scheduling on a tightly-coupled system such as a cluster or a massively parallel processing machine is based on the assumptions outlined by Berman[5]:

1. All resources are available at any time and can be scheduled accordingly
2. We have the authority to use every available resource
3. Inter-PE (processing element) communications are possible at any time with a predictable level of performance
4. The set of shared resources remains constant (unchanged)
5. There is minimal contention of resources among applications as all usage of the system is under the control of a central policy.

In a grid computing environment, all of these assumptions break down. Therefore, a scheduler for a grid environment has to be designed to take into account the following:

1. Resources may become available or unavailable at any time
2. Permission is required to schedule jobs across administrative domains
3. Direct communications may not be possible between all PE nodes, and when available the latency is higher and bandwidth is lower than what is typically available on a LAN or in a cluster.
4. The size of the resource pool can expand or contract
5. There are numerous applications all wanting to take advantage of the best resources. Not all these resources are under control of a centralised scheduling policy, therefore

cooperation is required between schedulers at different virtual organisations.

3.3.4 Decentralisation

Grids by their very nature are decentralised. Grids are formed by multiple parties cooperating together to share resources, but since these parties are also interested in autonomously administrating their computing facilities there is no centralised authority that dictates how this sharing should occur. Thus, sharing is based on mutual agreement, and each participant has the freedom to decide what it wants to share, when they want to share it, and how much of it they want to share [23].

Therefore, there is a need for decentralisation in the design of services that will be used to construct a grid. Iamnitchi and Foster found that a decentralised and self-configuring architecture is a promising solution for dealing with a large collection of dynamic collection of distributed resources [33]. Iamnitchi and Foster also noted the similarity between grids and other decentralised computing systems such as peer-to-peer file sharing [20]. The main similarity of interest in their paper is that the decentralised nature of both types of systems allow them to scale well over the internet with a large number of users. Peer-to-peer systems have been developed as specialised solutions for specific resource sharing problems (addressing anonymity, scalability and availability), and thus both technologies may converge in the future.

3.3.5 Residual Dependencies

A residual dependency is process state information left on the home node that is still required for the process to function correctly after it has migrated away to another host. The implementation of distributed processes with the design goal of allowing as much disconnected operation as possible means that residual dependencies must be minimised.

However, it is difficult to eliminate them all. Residual dependencies occur in the form of open socket connections and other IPC mechanisms, open files, unused virtual memory pages (for the pre-copying and copy-on-reference migration strategies), devices in use, process identification information and other things specific to different operating systems. Douglis and Ousterhout state that residual dependencies are undesirable because they decrease reliability and performance while increasing the complexity of the distributed system [13].

3.4 Wide Area Process Migration

Central to the development of Zinc is the migration of processes across a wide area network. Unlike the migration of processes in Condor which occurs in a cluster or a LAN, wide area process migration can occur between LAN's or across the internet. There are a number of problems which arise from this kind of distributed computing to be done transparently, as identified by Homburg *et al.* [31] as follows:

1. A very large number of users and resources
2. Latency due to distances between nodes
3. Heterogeneity of underlying operating systems and architectures
4. Involvement of different administrative domains

To handle process migration over a wide area network effectively, the distributed system must be constructed to handle migration and process I/O despite the problems above. When faced with a large number of resources, the system must be able to quickly identify and select resources based on its quality. This resource selection algorithm must be scalable

enough to handle a large number of resources. This is handled by the scheduler and the scheduling algorithms.

Perhaps the biggest obstacle for distributed computing is the latency of intercommunication between nodes because of the distance between them. The greater the distance, the worse the latency becomes. Again, the scheduler has to be intelligent about migration and placement, by avoiding or minimising I/O across high-latency links. A filesystem which allows for disconnected operations also helps in this respect. However, since the scope of this research does not cover filesystems or inter-process communication, Zinc assumes no IPC takes place, and simplifies file access operations by simply packing the files in use by the checkpointed program together with the checkpoint.

As for heterogeneous systems, it is not reasonable to expect binaries to work unmodified across operating systems or architectures. There have been efforts to create virtualisation layers which allow for this, but for the purpose of this thesis we wish to migrate native binaries, therefore the operating system and architecture must be adequately uniform, or homogeneous. The architecture chosen is Intel IA32 (sometimes referred to as i386) and the operating system is Linux. This does not hinder our goal of a distributed and widely available network of computing, since these are very popular computing platforms.

The problem of migrating processes across administrative domains imply issues of priority and security. The owner of an administrative domain would not want a foreign process to use too much CPU power or disk space when it is scarce, and the concern of malicious foreign processes is always on the mind of a system administrator. There is also the issue of data integrity on foreign nodes; what if data is corrupted or subverted on a remote node? In the development of this system, certain assumptions were made to simplify the process of inter-administrative domain cooperation and the security issues that it entails:

1. Cooperation between administrative domains is agreed on beforehand by the parties involved. Two parties that have agreed to cooperate mutually trust each other.

2. Trust is commutative; if administrative domain A trusts domain B, and domain B trusts domain C, then it is assumed domain A trusts domain C without the need for an agreement.

3. All transfer of data and programs is encrypted via ssh.

4. All executing programs are run with a specified Unix user and group ID for grid computing purposes. This user and group ID do not have access permissions to other parts of the system.

There are better methods of establishing trust and security (see section 7.5 on future work), but these choices were made to be as simple as possible, for security is not the primary focus of this work.

3.5 Performance Metrics

The performance metric when discussing distributed systems in literature has almost always been the mean response time of jobs [69]. Other metrics may include process wait time (the amount of time a process spends waiting for resources) and wait ratio (wait time per unit of service) [39]. However, one of the goals of grid computing is for gaining high throughput when utilising available resources [49]. In this respect, it would be more useful to look at *system throughput* instead of mean response time. The definition of throughput varies from publication to publication, but generally the accepted meaning is a measurement of the amount of work accomplished by a system in a certain amount of time. Thus, our primary performance metric when assessing Zinc is system throughput. For the purpose of our experiments, the throughput benchmark is the number of processes completed

by a particular administrative domain in one hour. The details on the experiments are discussed in Chapter 6.

3.6 Summary

The process of adapting an existing operating system to support grid extensions requires an overhaul of many of the major subsystems of the OS. However for our implementation we have chosen to implement only the process management aspect. Our goals in this implementation of grid process management are transparency, minimising residual dependencies, improved throughput, adaptability to changing resource availability and decentralisation. When handling wide-area process migration, we have chosen not to implement inter-process communication, and file I/O is handled on the local filesystem. When evaluating the system, our chosen performance metric is throughput instead of mean response time.

CHAPTER 4

DESIGN DETAILS AND METHODOLOGY

4.1 Introduction

In this chapter, we discuss the system structure of Zinc and the design details of the framework. In section 4.2 we take a look at how a Zinc system looks like. Section 4.3 is on the information collected and used for measuring load. Section 4.4 looks at distributed process management, namely how global process IDs are implemented as well as how a process is selected for migration. Section 4.5 describes how resource discovery, system selection and job execution is done in Zinc. In section 4.6, we look at the set of programs that we will use as control for our experiments. Finally, section 4.7 provides a summary for this chapter.

4.2 System Structure

In this section, the structure of the grid which is explored. Each participant uses the Zinc software to create the grid. The resulting system is greater than the sum of its parts, hence the usefulness of a grid is only seen when many different administrative domains participate.

4.2.1 Architecture Of The Zinc-based Grid

Figure 4.1 shows how a computational grid can emerge from the cooperation of several administrative domains. Each administrative domain can be considered a vertex in a

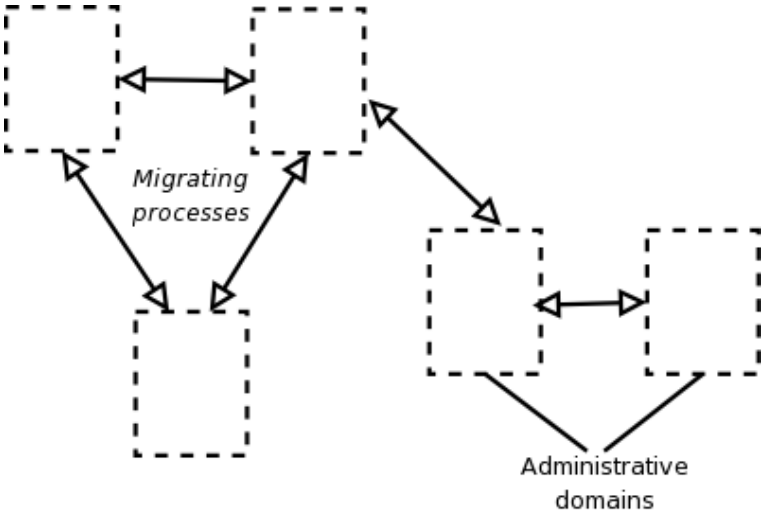

Figure 4.1: Administrative Domains On The Grid

graph, and the directed edges between the vertices represent the administrative domains being fully operational and connected to the others, and ready to participate in the grid.

Initially, any participant of this grid do not have to know the status of every other participant, therefore we may start with an incomplete graph. The Name Dropper resource discovery algorithm (described in chapter 2) will ensure complete graph connectivity once Zinc runs for a while. Therefore, it is possible for administrative domains to join a grid that's already fully connected by simply announcing itself to one of the other administrative domains, and the others will pick up its presence quickly and begin sending jobs to and accepting jobs from that domain.

Processes have the ability to migrate around different administrative domains, thus enabling them to make use of any resources that might be available throughout the entire interconnected grid. These features work toward our goal of an operating system adaptable to a dynamic grid (where administrative domains can come online arbitrarily and is accommodated quickly).

4.2.2 Administrative Domain Interaction

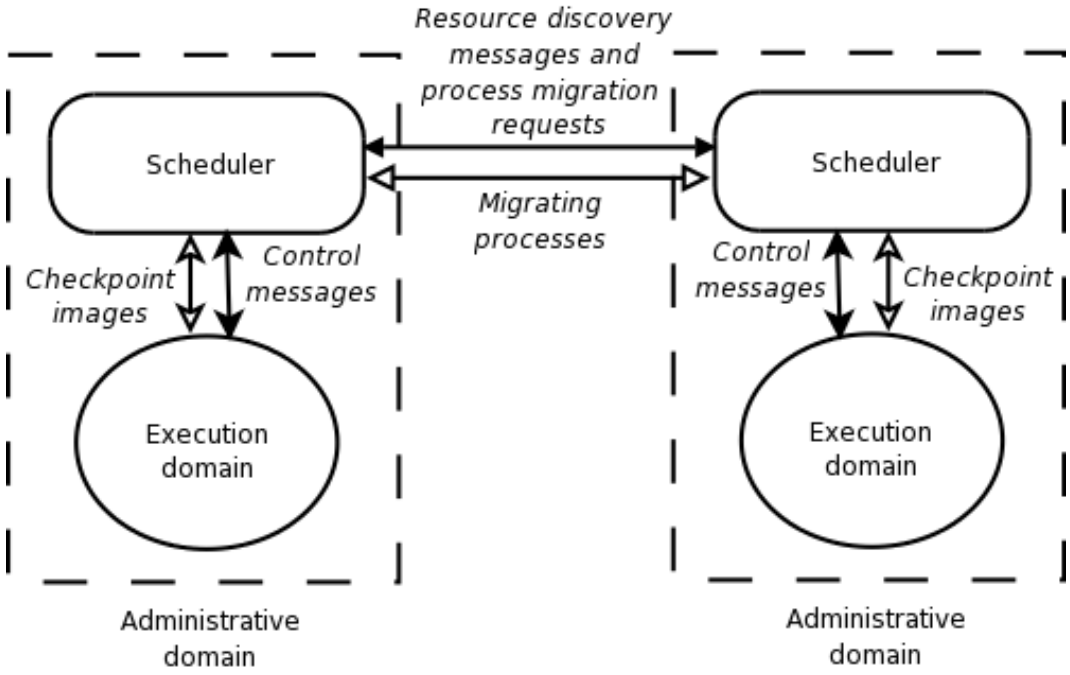

Figure 4.2: Administrative Domain Interaction Details

Between administrative domains, there is a lot of intercommunication between the schedulers that oversee the execution of processes in each administrative domain (figure 4.2). These schedulers pass the following information back and forth when required:

1. Process migration requests
2. A transfer of process checkpoints (together with whatever data files the program was using)
3. Resource discovery messages

Administrative domains are peers, and thus behave in a decentralised manner without a central authority to control them. This is a necessary design element based on our goal to have a decentralised grid where administrative domains retain full control of their own resources and sharing only to the extent the local policy permits.

4.2.3 The Execution Domain And Controller

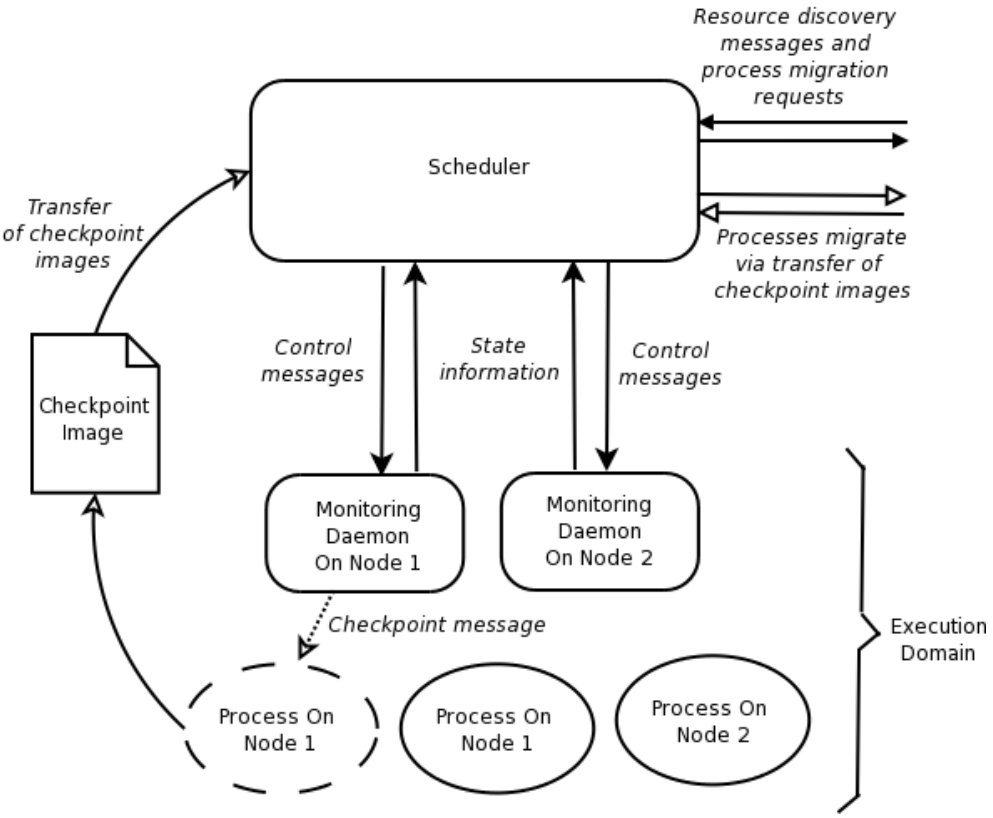

Figure 4.3: The Scheduler And Its Execution Domain

The execution domain is a set of n computing nodes (where $n \geq 1$) which is under a particular administrative domain which processes can execute on. The processes may be migrated to any other node in the same execution domain without the need for inter-scheduler communication. Each execution domain is governed by one scheduler, running on a dedicated server which acts as a controlling node. Each node will run an execution and monitoring daemon which records execution information and passes it to the scheduler. Finally, each node runs on a modified Linux kernel that allows checkpointing of individuals processes. The software components that comprise Zinc are described as follows:

1. The EPCKPT-enabled Linux kernel which exports a checkpointing system call to userspace which can be called to checkpoint a process into a checkpoint file.

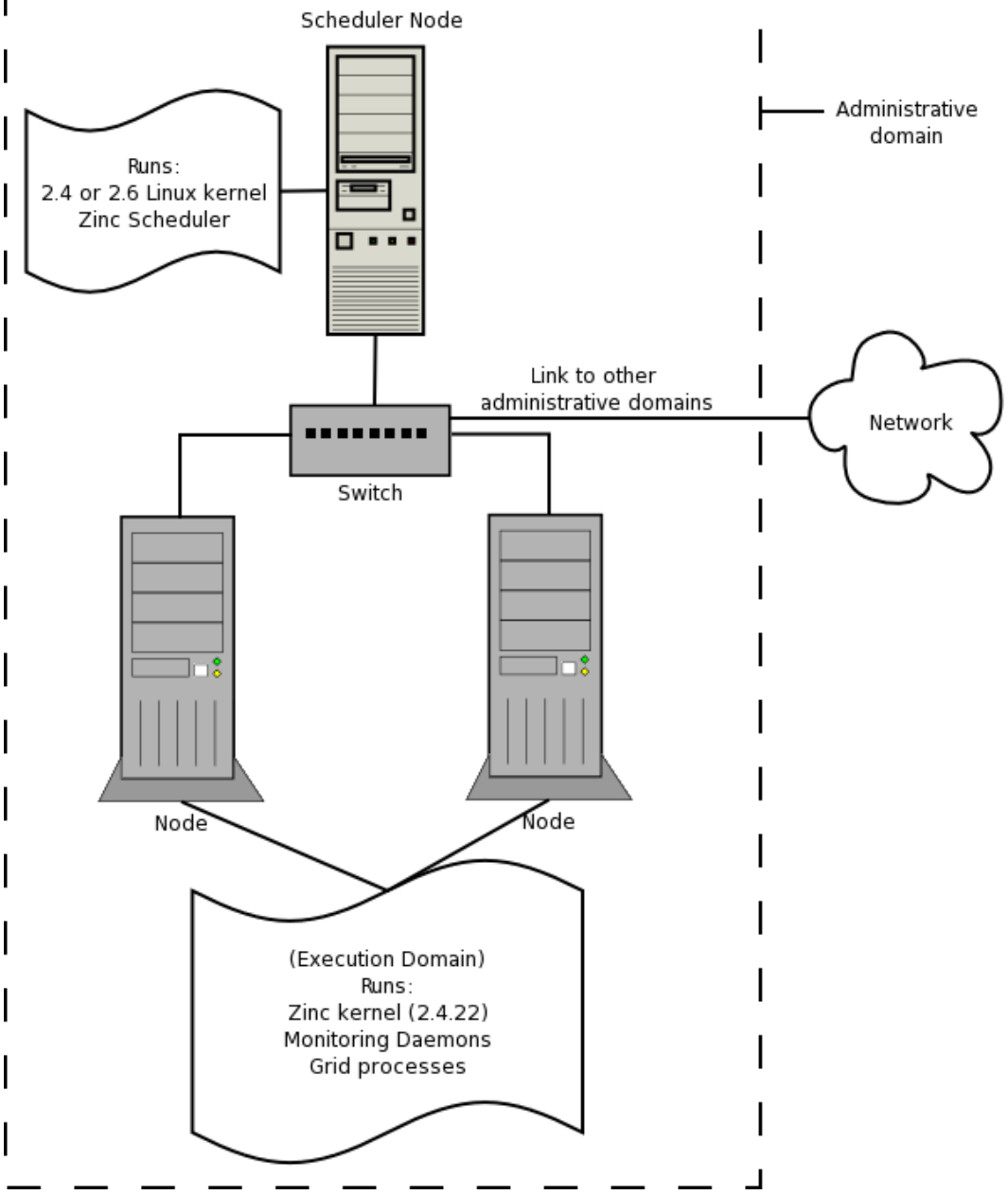

Figure 4.4: Mapping An Administrative Domain Onto A Physical System

2. The Zinc userspace scheduler is the entity which evaluates the load conditions of its execution domain, and based on this information it will make scheduling decisions at every interval of ϕ seconds (where ϕ was arbitrarily chosen at 20 for the implementation). The criteria for these decisions are discussed in the following section.

3. An execution and monitoring daemon does the actual movement and placement of grid processes and process checkpoint images. It relies on the `sshd` program to

transfer checkpoint images, and passes messages back and forth with the scheduler via TCP sockets.

Figure 4.4 shows how an administrative domain maps onto a sample set of hardware where there is one scheduler and two execution domain nodes. Figure 4.3 shows the same administrative domain with interaction between the software components. Figure 4.5 shows how the kernel and monitoring daemon uses it to checkpoint processes. The implementation of these software components are described in the next chapter. For implementation details of the checkpointing kernel modifications see section 5.3, while the scheduler implementations (both kernel and userspace) are discussed in section 5.4.

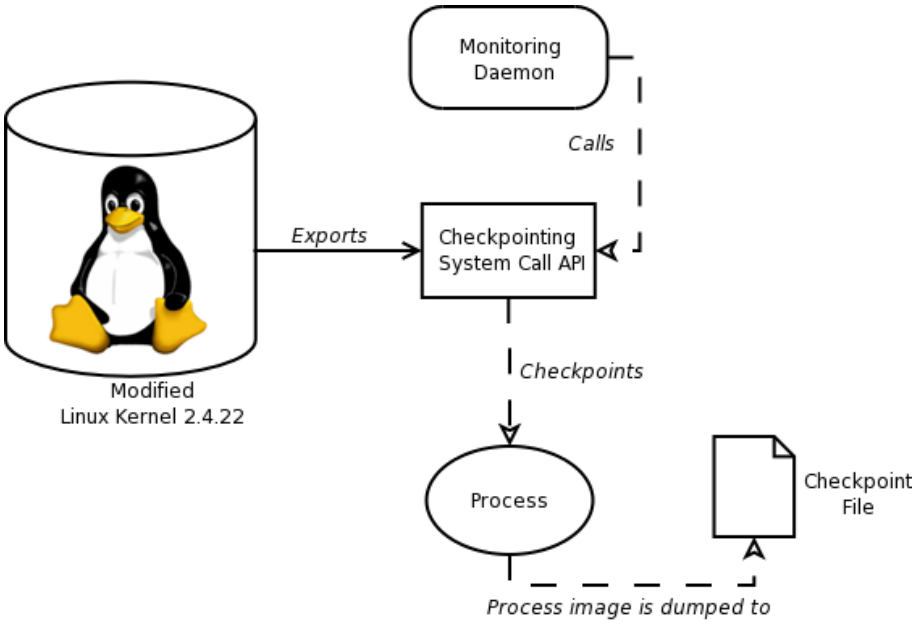

Figure 4.5: The Kernel And Monitoring Daemon

4.3 System Information

4.3.1 Processor Load

System load is the number of all processes (grid processes or otherwise) in the system run queue in the past 1 minute. Run queue length is an effective metric for determining system load, as determined by Ferrari and Zhou [19], and is provided for by the Linux kernel. Thus we have adopted this as an index for processor load.

4.3.2 Threshold

One of the decisions we had to make was how to decide just how "overloaded" a computing node was before a decision is made to evict grid processes from it. Our threshold policy is a function of CPU "strength" ($bogomips_x$)and installed memory size ($memory_x$), which serves to represent the computing capabilities of a particular node x:

$$T_x = (tanh(\frac{bogomips_x}{4000}))(tanh(\frac{memory_x}{1000}))(10) \tag{4.1}$$

It was found from our observations of load behaviour on the equipment we used that for the most powerful of nodes, a load level of 10 was enough to make it slow and unresponsive. Thus, based on this observation, equation 4.1 guarantees that the value of T_x is always less than 10. This function is not based on any theoretical threshold calculation, but was manually adjusted until it produced appropriate thresholds for every node in the grid test-bed.

We used "bogomips" [1] of the node's processor to get an estimate of how fast it is.

[1] Bogomips or "bogus millions of instructions per second" isn't really a measure of computing capacity. It is used by the kernel for timing calibration by counting the number of busy loops it can execute in a particular period of time. However, we found that bogomips can be a useful (albeit very crude) approximation of CPU power.

Based on the above function, the threshold value increases when we have more powerful CPUs and more memory, hence allowing the system to accommodate higher loads before migration is initiated. The hyperbolic tangent of the parameters in the function effectively normalises it between 0 and 1. The values 4000 and 1000 and 10 are constants used as weights in the equation until the threshold function "works" (adequately sets a threshold for the node that prevents system overload) for all the computers in our grid test-bed, calibrated manually by observing the performance responsiveness of different nodes under varying levels of load for the machines that were used. Further research is still required to accurately determine optimal thresholds without resorting to manual calibration.

4.4 Managing Distributed Processes

4.4.1 Global Process IDs

The challenge of managing processes on a wide-area distributed network is keeping track of the location of each process, and tracking information about them. Linux (and all Unix-like operating systems in general) track processes running on a local machine via it's process id, or pid. On the i386 architecture, the pid is defined as an integer, but the maximum allowed processes on a Linux 2.4 system is 32,767 (a 16-bit value). This is to maintain compatibility with older 16-bit Unix systems which enforced such a limit.

This arrangement works well enough on a single computer, where each process can be accounted for to be either running, sleeping (there are two kinds of sleeping states : interruptable and uninterruptable), stopped or zombie. However, when one adds the possibility that a process may be checkpointed and migrated, we must add mechanisms to identify processes uniquely across machines, and also to track their location.

Since Linux does not have a distributed design (each machine runs its own kernel), it is

difficult and impractical to provide unique kernel pid's for all processes in the distributed system. Here are some of the issues faced:

1. There need to be an agreement to which nodes are allowed to use which pid numbers which would be impossible to do for a system where administrative domains are allowed to join and part at any time.

2. The pool of usable pid's are relatively small, and increasing the number would break compatibility with some applications.

3. A need to introduce new states into the kernel, and these states would need to be synchronised with remote nodes when migration occurs

These problems are not easily solved in-kernel. Attempting to do so would needlessly bloat the size and complexity of the kernel. Therefore, another layer of identification was added to userspace. Each process in the distributed system is uniquely identified by:

1. The IP address of the zinc userspace scheduler in charge of the administrative domain (which has a unique public IP)

2. A set of positive integers called `uniqueid` assigned in sequence to processes started on any machine in the administrative domain. Each number in this set is unique. This number is assigned to the process by the central zinc userspace scheduler of the administrative domain, and the scheduler maintains a mapping of `uniqueid` to local pid for each grid process spawned within its domain.

The Zinc userspace schedulers will only schedule and manage processes started as "grid processes", via the `gstart` command which submits the process to the system. Thus, all

"grid processes", marked as such by Zinc, will get a global ID which combines the IP address of its administrative domain and its `uniqueid`.

4.4.2 Selecting Processes For Migration

When selecting a process for migration, we employed a relatively simple algorithm which has the goal of reducing the load on a particular host. Most process selection policies in previous work are functions of process age, where a process that was "old enough" are selected for migration [28, 44, 40, 2]. We developed our process selection policy based on the following criteria:

1. Only grid processes are migrated, thus reducing the set of eligible processes to ones that have a high probability of being CPU-intensive.

2. We used CPU cycles used by a process to represent its age, rather than how long it has been since it started. Processes that use up more cycles tend to be heavy CPU users that contribute to the load. The more cycles the process has used (the older it is), the more likely it's going to be migrated.

3. Processes that use more memory will tend to be migrated, freeing up memory on the loaded node, which increases its performance. Despite the cost of moving bigger checkpoint images due to big virtual memory sizes, it is still worth doing for the performance improvements on the host node when such processes are migrated away.

4. Processes which have already been migrated should be less likely to migrate again. Each time a process migrates, its likelihood of being migrated becomes less.

Thus, the ranking algorithm in Zinc evaluates all grid processes in the system, and ranks each process, p according to the following function which reflects the requirements of the

above criteria:

$$Rank_p = \frac{CPU_p * VMSize_p}{Checkpoints_p + 1} \tag{4.2}$$

CPU_p is the number of CPU cycles used by process p, $VMSize_p$ is the virtual memory footprint size of p, and $Checkpoints_p$ is the number of times the process has been checkpointed. The value of 1 is added to the denominator because we do not want a division by zero condition when the process has never been checkpointed. Therefore, the more CPU or RAM the processes uses, the higher ranked it is in the list of running grid processes. The more times it has been migrated, its rank grows less.

If at a scheduling point, a given host is past the load threshold, the scheduler will pick the highest ranked process based on the above algorithm, and have it migrated away. The scheduler will continue to keep ranking and picking a process for migration every scheduling interval until the load is reduced below the threshold.

4.5 Scheduling

The three major components to grid software encompass the functionality of resource discovery, resource allocation and execution monitoring [67]. Zinc implements a basic subset of the three stages, and is described in the following subsections.

4.5.1 Name Dropper : Resource Discovery

Each administrative domain tracks the following information about every node n in its execution domain by polling them every 20 seconds:

1. Hostname, h – a unique network identifier for the node within the administrative domain
2. Processor load, l, on the node,
3. Available memory, m, on the node
4. Timestamp, t – when the information was obtained.
5. Threshold, T, for the node
6. Memory usage ratio, r – what percentage of memory has been used by the system

This information is collectively referred to as *node state*, ν. The node state for node q at time of polling t is:

$$\nu_q^t = \{h_q, l_q, m_q, t, T_q, r_q\} \tag{4.3}$$

The *execution domain state*, ϵ is the set of all node states in an administrative domain. The execution domain state of administrative domain u with n nodes at time of polling t is:

$$\epsilon_u^t = \{\nu_1^{t+\delta 1}, \nu_2^{t+\delta 2}, \ldots, \nu_n^{t+\delta n}\} \tag{4.4}$$

Since the polling isn't done synchronously, there are slight differences between timestamp values for each node (hence, there is a small difference of δ between polling time t and actual timestamp obtained for each node). However, because all the nodes in ϵ_u are polled at once on a low-latency, high-speed internal network, the timestamp values of each node are more or less equal, and the values of each δ should be negligibly small. For the Name Dropper algorithm to work, the largest value of δ for all nodes in an administrative

domain should be less than the time between rounds ϕ. However, if the number of nodes increases, the value of the largest δ will also increase, in which case a bigger ϕ should be chosen.

Thus, the administrative domain state for u at time t can be described by:

1. The IP address of the scheduler, IP_u
2. Its execution domain state, ϵ_u^t

The resource discovery algorithm is used to announce the presence of administrative domains, and also to disseminate the above state information and propagate it throughout the system. The algorithm works in the same way as the standard Name Dropper, with minor changes to accommodate the extra information that needs to be transmitted to other nodes. The *neighbour list*, $\Gamma(u)$ in the modified Name Dropper not only includes the identifiers of neighbouring (and its own) administrative domains (in this case, the IP address of the scheduler) but also the execution domain state at time t of the neighbours (and likewise, its own execution domain state). Therefore, its neighbour list at time t is denoted as $\Gamma(u)^t$.

At each round, each node u transmits $\Gamma(u)^{t_i}$ to *one* randomly chosen node v where $v \subset \Gamma(u)^{t_i}$. Upon receiving $\Gamma(u)^{t_i}$, node v will update its own neighbour list $\Gamma(v)^{t_j}$ with new information from u. For information about administrative domains that already exist in $\Gamma(v)^{t_j}$, the new information is updated if and only if $t_i > t_j$.

Having the same efficiency as the standard Name Dropper algorithm, each administrative domain will learn the system state very quickly in $O(log^2 n))$ rounds with high probability.

4.5.2 Zinctask : System Selection

User of the Zinc grid will submit their jobs at the administrative domains that they belong to (i.e., have a login and password). They can either submit their job to the scheduler, where the scheduler will randomly select a node in the execution domain where it will run, or the user can explicitly specify which node they want to initially run the job on. Initial placement does not matter much from the system's point of view. Only when the load of a particular node q exceeds its threshold value, $l_q > T_q$, does the scheduler select another node to migrate grid processes to.

The scheduler maintains two tables, one is a table of all grid processes submitted to it, and another is a table of all computational nodes available for execution in the zinc grid, extracted from Γ. The indices for the process table, P is the list of n processes, p with `uniqueid`, d paired with the IP of the scheduler it belongs to, IP:

$$P = \{p_{IP1}^{d1}, p_{IP2}^{d2}, \ldots, p_{IPn}^{dn}\} \tag{4.5}$$

The execution nodes table, E is indexed by a list of n nodes, ν with hostnames, h paired with the IP of the scheduler it originated from, IP:

$$E = \{\nu_{IP1}^{h1}, \nu_{IP2}^{h2}, \ldots, \nu_{IPn}^{hn}\} \tag{4.6}$$

When a process p is chosen for migration, it is marked for migration and the host on which to migrate it to is decided upon by constructing a pair of *eligibility lists*, one being a list of all eligible nodes in the local administrative domain (e_{local}) and another is a list of all eligible nodes in all foreign administrative domains ($e_{foreign}$).

The nodes are accepted from E into the eligibility lists if:

1. For all local nodes:
    - The load/threshold ratio is less than 1 for the next 20 seconds (the duration of the scheduling interval)
    - The memory usage ratio is less than 0.66[2]
2. For all foreign nodes:
    - The memory usage ratio of less than 0.66
    - The connection uplink between the home administrative domain and the foreign administrative domain has a bandwidth of at least 48 kilobits/sec and a minimum latency of 10 seconds

When choosing a node within the eligibility list, all nodes in e_{local} and $e_{foreign}$ are ranked with a composite numerical value according to desirability. The following criteria were taken into consideration when designing the ranking algorithm:

1. All ranks are capped at a maximum value to prevent overflows when invoking mathematical calculations on a computer
2. Migration inside ones own administrative domain is more efficient than outside it, therefore a bias was introduced to encourage local migration: the maximum rank of foreign nodes is capped at 100 whereas for local nodes, the cap is 1000. Also, a local node should be higher ranked than a foreign node if the load is equal.

[2]The value 0.66 or 2/3 is an approximation of the amount of load the virtual memory system of Linux can handle before it starts swapping. It is an arbitrary value suggested by the Linux kernel programmer community, and does not really have any empirical justification besides "experience" by programmers in the field.

3. When evaluating load, one is more concerned about load in the next few seconds (when the process will arrive at the destination) rather than the load at the point of scheduling. Yang *et al.* [83] assert that such predictions help grid computing system attain better performance. Hence, a simple prediction mechanism was devised for local nodes. The scheduler keeps a load history of all local and foreign nodes. The local nodes' load history is recorded at every scheduling point. To predict the next load, we used a variant of the LAST prediction model [45], but instead of taking the last value, we construct a linear extrapolation function $f(t_{n+1})$ based on the last two entries of load l (out of n entries) of history data, when available:

$$f(t_{n+1}) = \begin{cases} 0 & \text{if } n = 0 \\ l_n & \text{if } n = 1 \\ \frac{l_n - l_{n-1}}{t_n - t_{n-1}}(t_{n+1} - t_{n-1}) + l_{n-1} & \text{if } n > 1 \end{cases} \tag{4.7}$$

   Although the actual load signal may be nonlinear, we found linear extrapolation to be an adequate approximate fit for the data at sufficiently small intervals in time.

4. The prediction algorithm does not work well for foreign nodes where unlike the regular polling of local nodes, we don't get data at consistent intervals for the foreign ones. Thus another factor, *reliability*, is introduced to penalise the ranking of any foreign node that has fluctuating load, by using a function of the standard deviation, σ of load history normalised to a value where $0 \leq \sigma \leq 1$:

$$reliability = 1 - tanh(\frac{\sigma}{2}) \tag{4.8}$$

The Zinctask algorithm is run to choose a host for the process to migrate to. The algorithm is shown in figure 4.6.

Once a host is chosen, the process needs to be migrated there. If the node is in the local administrative domain (ν_{local}), the scheduler simply migrates it. If the node

resides in a foreign administrative domain ($\nu_{foreign}$), a negotiation message is sent to that administrative domain containing the following:

1. The requesting domain's IP address
2. The `uniqueid` of the process that wants to be transferred
3. The virtual memory footprint size of the process
4. The node in the target administrative domain the process wants to run on

for all x in e_{local} **do**
  $capacity \leftarrow \frac{PredictedLoad(x)}{Threshold(x)}$
  $Rank(x) \leftarrow \frac{1}{capacity+0.001}$
end for
for all y in $e_{foreign}$ **do**
  $reliability \leftarrow 1 - tanh(\frac{\sigma}{2})$
  $Rank(y) \leftarrow \frac{reliability}{lastload(y)+0.01}$
end for
if highest rank of $x \geq$ highest rank of y **then**
  Return the highest ranked x
else
  Return the highest ranked y
end if

Figure 4.6: The Zinctask algorithm

Upon receiving this request, the target administrative domain compares the available memory on the target node, m with the virtual memory footprint size of the requesting process, v_{size}. If $m > v_{size}$, an acceptance message is given to the originating administrative domain and migration begins. However, if $m < v_{size}$, a rejection message is sent and the originating administrative domain removes $\nu_{foreign}$ from the eligibility list for one round and the host selection algorithm is begun again. The search continues until an accepting host is found. If the search exhausts all possible nodes, the process chosen to be migrated is unmarked and continues running.

It is anticipated that the Zinctask algorithm provides better job throughput than no placement strategy at all, because of the strategic and opportunistic placement of the "best" host according to the design decisions above will ensure processes have to share less CPU time amongst themselves if distributed wisely, thus getting more work done.

4.5.3 Zincd : Job Execution And Monitoring

The execution of jobs on a Zinc system is rather straightforward. A user submits a job to the grid much like he or she executes a regular program, but by using the `gstart` command. The user in an administrative domain can either specify a node in the execution domain which he or she wants the job to run on, or if unspecified, Zinc will randomly choose one for him or her.

The monitoring of processes running on a given administrative domain is done at every 20 second interval by the scheduler when it polls its execution domain. A message is sent out to every node, asking them to update the status information of the processes.

Once a process is completed, the completion time is logged and if its a foreign process, the home administrative domain is notified and the process with its data files are packaged and sent back to the originating administrative domain.

4.6 Control Software

In addition to the Zinc programs, a set of identical control programs were made that duplicates the functionality of Zinc, but without the checkpointing feature. The control programs use non-preemptive migration which use the same Zinctask algorithm for host selection. A job submitted to the control program is either run on a random node or a node of the user's choosing, just like Zinc. However, when a particular chosen node's load

exceeds its threshold, the Zinctask algorithm is invoked to choose a new host for running the job. If the Zinctask algorithms exhausts its eligibility list, the job is marked as "failed" and is not started. This is a necessary safety measure because the system is not able to migrate or checkpoint a job if it's already running, and may stall the machine if it attempts to run it anyway on an overloaded node.

During experimentation, artificial programs are used to represent CPU-intensive processes. To simulate jobs entering the system, these artificial programs performed a busy-loop with a counter (see figure 4.7).

```
int i,j,k,x;

for(i=1;i<=MAX;i++)
{
        for(j=1;j<=MAX;j++)
        {
                for(k=1;k<=MAX;k++)
                {
                        x = (i*j)/(i*j);
                }
        }
}
```

Figure 4.7: Code to generate a CPU-intensive process

The duration of how long the program runs on a particular node is adjusted by modifying the value of the symbolic integer constant `MAX`.

For tests that require a variable-sized memory footprint, a `malloc()` initialisation routine is added (see figure 4.8). The size of the desired memory footprint is adjusted using the `MAXSIZE` symbolic integer constant.

```
int * bigwhoop;

bigwhoop = (int *)malloc(MAXSIZE * sizeof(int));

if(bigwhoop==NULL)
{
        fprintf(stderr,"Malloc␣failed\n");
        exit(1);
}

for(i=0;i<MAXSIZE;i++)
{
        bigwhoop[i] = 1;
}
```

Figure 4.8: Code to generate programs with different memory footprint sizes

4.7 Summary

In this chapter, the design of the structure and algorithms for Zinc were presented. Although there are numerous issues in designing a distributed system, we have chosen the subset presented here to be the focus of our implementation. Global process IDs are handled by the scheduler, as well as resource discovery, process migration selection and placement. Besides the Zinctask placement algorithm we used in Zinc, we also implemented a random placement strategy, as well as a version of the Zinc userspace scheduler with process migration disabled.

CHAPTER 5

IMPLEMENTATION

5.1 Introduction

In this chapter, we discuss the implementation of the Zinc program and kernel extensions. Section 5.2 is a general introduction to the components of the software, followed by implementation details of checkpointing that is provided by the EPCKPT patch which we use for our system in section 5.3. Next, the scheduling mechanisms are discussed in section 5.4, both the kernelspace scheduler and the userspace Zinc administrative domain scheduler. Section 5.5 provides details on the resource discovery implementation and how Zinc extracts state information and monitors the system. The command line tools we have developed for interacting with the system is given a brief mention in 5.6 before a summary in section 5.7.

5.2 Implementation Overview

The goal of the implementation of Zinc, is to add a layer of systems software on top of GNU/Linux to make it "grid-enabled". GNU/Linux is not a distributed operating system, nor can it be made a full-blown distributed operating system without considerable redesign and rewrite of the kernel. However, our goal is to provide a "grid-aware" kernel and userspace, so that processes may be moved about across the Zinc-enabled grid with ease. To achieve that goal, we need to minimise the amount of intrusive changes introduced into the Linux kernel, by only adding functionality that cannot be done outside of kernelspace. Therefore, the implementation for Zinc consists of 2 parts: kernel-side code and userspace

code.

1. The kernel-side code consists of a Linux 2.4.22 kernel containing the following updates and modifications:

    - EPCKPT [61] is code initially written by Eduardo Pinheiro that adds checkpoint and restore functionality to the Linux kernel.
    - A process scheduler optimised for fairly scheduling compute-intensive jobs. This scheduler was our collaborative development effort with the Linux kernel developer community, and was developed mainly by Constantine Kolivas [32], based on the kernel scheduler of Ingo Molnar [54].

2. A sample implementation of the grid services which will utilise the underlying operating system "hooks":

    - A distributed grid process scheduler in userspace (the Zinc userspace scheduler)
    - A daemon for tracking and monitoring of running grid processes (the Zinc monitoring daemon)
    - A utility for launching grid-aware processes (the `gstart` command)

The system as a whole would provides a computer with the tools to connect to other similarly equipped computers, effectively sharing resources across the entire distributed system. Processes may be migrated across this distributed system and use the aforementioned available resources.

A user logging on to any particular node in this distributed system does not see a single system image, but instead is presented with a process execution service which may be carried out on any machine in the grid.

The entire grid would run with each node having it's own kernel, like a network operating system. There would be too much kernel bloat if direct inter-kernel communication were allowed and the grid scheduling took place in-kernel. Therefore, the kernel only supplies the bare minimum necessary to implement process management; by allowing processes to be checkpointed and restarted elsewhere. The kernel also implements a scheduling policy that allows for another userspace scheduler to handle job priority reliably. The details of the mechanisms that are implemented in this system is described in the following sections.

The following sections will discuss the implementation of each component of the system in detail. When referring to the Linux kernel, the reference is specific to the 2.4.22 version which was used in the implementation. Later (or earlier) versions of the kernel may differ in implementation details.

5.3 Process Checkpointing

Process checkpointing is one of the strategies for process migration. A running process can be checkpointed into a file that contains the state and instructions of the process. By transferring this file to another machine and restarting it, we achieve process migration. It is necessary to implement process checkpointing in kernelspace, as the state of a running process exists both in kernelspace and userspace. Linux in its current form does not officially support process checkpointing. However, there have been a number of projects that add support for this functionality into the Linux kernel, with varying levels of success. We needed only to provide a checkpointing mechanism for Zinc, thus we chose a simple patchset already implemented for Linux that does this with minimal changes.

EPCKPT[61], in its initial form was a kernel patch architecture developed by Eduardo Pinheiro to provide explicit checkpoint and restore functionality to Linux. It was originally

designed to work with early versions of the Linux kernel (2.0, 2.2 and early 2.4). The functionality provided was:

1. Support for the i386 architecture (process checkpointing support is architecture-specific).
2. Checkpoint of any running process into a checkpoint file.
3. The checkpoint file is an executable that resembles ELF hence the resuming of program counter and restoring of memory map and program state can be done by executing the checkpoint. This is achieved by adding a binary format handler to load checkpoint files as executables in `fs/binfmt_elf.c` in the Linux kernel source tree.
4. A system call interface to the checkpointing facility.

This functionality was decided to be the most flexible way to implement process checkpointing and migration, and thus was chosen to be used in the Zinc project. The version available was a kernel patch against 2.4.2, but we ported the patch forward to a more recent 2.4.22 kernel. We had to clean up a number of things, mainly eliminating side-effects due to the change in the kernel version. The core functionality however, remained the same. The major changes we made were incrementing the internal system call symbol to avoid conflicts with new system calls introduced in the interim versions of the kernel from 2.4.2 to 2.4.22. The second item we updated was the mechanism for reopening files was deferred to userspace, as the implementation in kernelspace in Pinheiro's initial patch was incomplete.

The checkpoint and restore mechanism is exposed to userspace via two system calls introduced in the EPCKPT patch:

1. `checkpoint (int pid, int fd, int flags)`

   This system call expects the pid of the process to be checkpointed as its first argument, a file descriptor to which to write the checkpoint image as its second argument, and optional flags to modify the checkpointing behaviour in the third argument. The checkpoint image writes to any file descriptor, which means it can write to a file or to a socket, making it convenient if one wants to migrate the checkpoint image right away. However, this migration feature is not used in Zinc as we create checkpoint images on disk first before migrating them. This is to accommodate the two-level scheduling hierarchy that does not allow execution nodes to communicate directly. The `flags` argument can take several options:

   - `CKPT_CHECK_ALL` Checkpoints all descendent processes of the process which will be checkpointed. This option is not used by the Zinc userspace scheduling algorithm, as single independent processes are assumed.
   - `CKPT_KILL` Kills the process after it is checkpointed. If this option is not specified, the program will continue running. This option is used by the Zinc userspace scheduler.
   - `CKPT_NO_SHARED_LIBRARIES` By default, the checkpoint will include both the executable image and any loaded shared libraries it uses. This is particularly useful when there is no guarantee that the host the process gets migrated to will have the same shared libraries. However on a software-homogeneous distributed system, this is unnecessary and can this option can be used to save some disk space. Zinc should be able to run on a system that has heterogeneous Linux installations, so this option is not used.
   - `CKPT_NO_BINARY_FILE` Does not include the executable file in the checkpoint, and relies on the system to have it available in its original location. This option

is not used by Zinc.

2. `collect_data (int pid)`

   Before a process can be checkpointed, the kernel needs to collect data about the process while it is in execution, such as the files it opened, children it forked off, and memory that it allocated. Therefore, any process that we know will be checkpointed needs to be marked with this system call on start up. This is acceptable, as to reduce memory overhead, we do not want to collect data about every process. The Zinc userspace scheduler deals exclusively with "grid processes" which will be marked as such with the `collect_data()` system call when `gstart` is invoked to start the process. This system call expects the pid of the process we want to collect data on as a parameter.

In the kernel, the information saved about any running process is stored in a struct called `task_struct`. The definition to `task_struct` in kernel 2.4.22 has fields for the following:

1. Process id (pid), as well as group and and user id's
2. Process state
3. A pointer to the process' address space
4. The open file descriptors of a process
5. Pointers to parent and child processes
6. Other maintenance information such as signal handlers, locks, ipc information etc.

The only fields EPCKPT is concerned with is the pointer to the address space and the pointer to the file descriptors used by the process. The other fields are simply discarded.

This is perfectly acceptable, since once a process restarts, the only information that needs to stay persistent in our design requirements is the address space, open file descriptors and their state, and the CPU registers.

Further information on the exact structure and function of the process descriptor, memory descriptor, file descriptor table, as well as the rest of the Linux kernel can be found in [6].

When a user (or a user program) wants to request for a particular program to be checkpointed, it calls the checkpoint system call. Inside the kernel, the `sys_checkpoint()` function will check if the file descriptor and process id given to it is valid. If the file descriptor and pid is valid, the process is sent a signal to indicate it is to be checkpointed. Eduardo Pinheiro used `SIGUNUSED` for this purpose. If the `CKPT_CHECK_ALL` flag is specified, it will find all children processes of pid and send the signal to them too. If the process requested for itself to be checkpointed, no signal is sent, but the kernel function `do_checkpoint()` is called immediately instead. The in-kernel signal handler for a process that gets the `SIGUNUSED` signal will also simply call the kernel function `do_checkpoint()`. The function takes one argument of type `struct pt_regs`, which is a struct that incorporates the current register state of the current process.

The `do_checkpoint()` function is where the actual checkpointing takes place in the kernel. The function will save the necessary state information to disk. The open file descriptors are described by the `struct files_struct *files` field in `task_struct`. Table 5.1 shows some of the fields in the `struct files_struct` structure which is used by `do_checkpoint()` during the checkpointing process.

First, the `file_lock` spinlock is set to ensure exclusive access to the field by the

Struct fields	Description
`rwlock_t file_lock`	Read/write spin lock for the data structure
`int max_fds`	Current maximum number of file objects
`struct file ** fd`	Pointer to array of file object pointers
`fd_set * open_fds`	Pointer to open file descriptors

Table 5.1: Fields representing open files of a process in `struct files_struct *files`

checkpointing function. Then, for each file object in `struct file ** fd` up to `max_fds`, the corresponding file descriptor in `open_fds` is checked to see if it's open. If it is, then the each file object's filename, access mode, flags and file pointer position are all dumped into the checkpoint file header. If the file descriptors are for pipes, then the pipe information is marked as such, and saved. EPCKPT however does not save the state of sockets, so file descriptors that represent open sockets are simply ignored.

Next, the checkpointer saves the process address space. The memory usage of a process is described by the `struct mm_struct *mm` field in `task_struct`, called the *memory descriptor*. The memory descriptor points to the process address space which is a complex data structure containing information about how the program running in the process is laid out in memory, plus additional information to help with the memory management for the process. The information that is of interest to the checkpointer is presented in Table 5.2.

It is fairly straightforward to save the values of all the virtual address delimitersm then traverse the list of memory region objects pointed to by `struct vm_area_struct *mmap` and write them out into the checkpoint file.

Lastly, the checkpoint function reads the values of the registers from the `struct pt_regs regs` (see figure 5.1) argument that was passed to it.

The entire contents of the structure is dumped to the checkpoint file, and thus com-

Struct fields	Description
`unsigned long start_code`, `unsigned long end_code`	Delimits the virtual addresses of the executable text
`unsigned long start_data`, `unsigned long end_data`	Delimits the virtual addresses of initialised data
`unsigned long start_brk`, `unsigned long break`	Delimits the virtual addresses of heap between initial virtual address and current end address
`unsigned long start_stack`	Initial address of user mode stack
`unsigned long arg_start`, `unsigned long arg_end`	Delimits the virtual addresses of where the command-line arguments are stored
`unsigned long env_start`, `unsigned long env_end`	Delimits the virtual addresses of where the environment variables are stored
`struct vm_area_struct *mmap`	Pointer to the head of the list of memory region objects

Table 5.2: Fields representing process address space in `struct mm_struct *mm`

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

Figure 5.1: The structure of `struct pt_regs` for the i386 architecture

pleting the checkpointing process.There are other state information in the process, but since IPC is not supported by EPCKPT nor is it necessary in our design assumptions, shared memory and other IPC mechanisms (such as sockets) are not saved. The checkpoint file is saved in a format that resembles ELF (executable and linking format) with additional headers, and EPCKPT adds support for this modified ELF as an executable format. Therefore, to restart the checkpoint, all the user needs to do is execute the file,

and a new process descriptor will be set up for it, with a new execution context, with the state information reconstructed from the information stored in the checkpoint file. The restart function in the kernel sets up all the necessary data structures and fills them out one by one based on the information read from the checkpoint. Any other maintenance information is recreated anew, including the process id. This is necessary because there is no guarantee that the pid number used by the checkpointed process will be available on the host it migrates to. However, the userspace scheduler maintains a unique id for all processes started as "grid processes", which allows the tracking of specific processes.

```
struct open_files
{
    int entry_size; /* How many bytes are we dumping after
       this struct */
    int type; /* CKPT_PIPE, CKPT_FILE, others */
    int fd; /* Original fd */
    union
    {
        struct
        {
            unsigned long inode; /* just a unique identifier
               of inode */
            unsigned rdwr:1; /* 0 - read, 1 - write */
        } pipes;
        struct
        {
            unsigned long int file_pos;
            unsigned long file; /* unique identifier of struct
                file */
            int flags;
            int mode;
            char *filename; /* see (1) */
        } file;
    } u;
};
```

Figure 5.2: The structure of `struct open_files`

The EPCKPT patch does not reopen files which were in use. To solve this, we decided to do this in userspace. We integrated the reopening of files into the `gstart` program that

is used to execute grid processes, thus ensuring that files will be reopened whenever a user starts or restarts a "grid process". The reopening of files in use by the `gstart` program is accomplished by parsing the checkpoint file for the list of file descriptors that were used to open files, and extracting the filename, access mode, flags and file pointer position into an array of `open_files` structures (called `openfiles`) as shown in figure 5.2.

Looping through the array using the variable `i` as a counter, the `gstart` program re-opens each file using the `open` system call (see figure 5.3).

```
filedes = open(openfiles[i].u.file.filename,
   openfiles[i].u.file.flags, openfiles[i].u.file.mode);
```

Figure 5.3: Calling the `open()` system call

Then, the file descriptor is set to a number matching the old one using the `dup2` system call (see figure 5.4).

```
ret = dup2(filedes, openfiles[i].fd);
```

Figure 5.4: Calling the `dup2()` system call

Next, the file position is set to the place last used when checkpointed using the `lseek` system call (see figure 5.5).

```
ret = lseek(filedes, openfiles[i].u.file.file_pos, SEEK_SET);
```

Figure 5.5: Calling the `lseek()` system call

Finally, `gstart` will call `execve` to restore the checkpointed program image, while

inheriting all the open file descriptors (which is the standard behaviour for `execve` on Linux).

5.4 The Schedulers

5.4.1 The Kernel Scheduler

The scheduler is the part of the operating system kernel that decides how a system is multitasked, by assigning each process on the system to a CPU for a specific time quantum, before it is preempted and another process is allowed to run. The time quantum given to a specific process is based on the Unix "nice" value that is assigned to that process by the user, and is then worked out into a priority value also based on other factors.

The default Linux 2.4 scheduler works as follows:

1. The scheduler maintains 1 queue of processes per CPU.
2. CPU time is divided into epochs. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
3. Whenever a process runs, it uses up its time quantum
4. An epoch ends when all processes have used up their time quanta. Then the kernel recomputes the time quanta of all processes, and another epoch begins.

This default scheduler is of $O(N)$ complexity, where N is the number of processes in the system.

At the time when Zinc was initiated, the next generation kernel (numbered 2.5.x) was still under heavy development[1]. It was during development of kernel 2.5.x that the kernel

[1]The development kernel has since been released as the stable 2.6.x series kernel, and at the time of

developer Ingo Molnar introduced a new scheduler into the Linux kernel, where scheduling decisions can be made in $O(1)$ time. The algorithm works as follows:

1. The scheduler maintains two types of processes queues per CPU, one for active processes and another for processes with expired timeslices. Each queue has sub-queues for each Unix "nice" priority value is associated with. Therefore, inserting a new process into a queue is also an $O(1)$ time operation, and the processes become effectively "sorted" by priority.

2. Every process of the equal priority X is given N units of timeslice, and each process in the active process queue is allowed to run for its allocated timeslice.

3. If a process has priority Y and $Y > X$ then it is given M units of timeslice where $M > N$.

4. Unless preempted by another process of a higher priority (if one wakes up), a process will run until it's timeslice expires or it goes to sleep voluntarily.

5. When a process's timeslice expires, it is removed from the active queue and inserted into the expired queue. Once the active queue becomes empty, the two queues swap; expired queue becomes the active queue and the active queue becomes the expired queue.

Scheduling algorithms for Linux use dynamic priority; the number of timeslices allocated to any particular process is a function of its Unix "nice" value and interactivity behaviour. The scheduler would tend to promote the priorities interactive processes higher to make them seem more responsive to user input. This behaviour is unnecessary for compute-intensive grid servers, and is even detrimental to the throughput of CPU-intensive

writing is at version 2.6.17.1

processes. Therefore, we collaborated with kernel developer Con Kolivas to produce a scheduler for Linux 2.4 based on the new development scheduler, using the Ingo Molnar's algorithm but with the following changes:

1. In-kernel priority is static and based entirely on the Unix "nice" value. This would ensure that each process is treated as fairly as possible based on the priority assigned to it by userspace. This enables the userspace scheduler to enforce priority policies instead of the kernel scheduler, which simply runs and schedules processes at the priority that was assigned to it and does not promote or demote the priority of that process.

2. Timeslice quanta are increased to allow processes to get more benefit from the CPU cache (the "hot cache" effect).

3. Even when processes are preempted (by another process of a higher priority value), it is delayed for a short time so that even lower priority tasks get to run for some time bound to the CPU before being expired to get hot cache benefits.

According to Hussein *et al.* [32], the Zinc kernel scheduler was benchmarked by to have a 40% throughput increase compared to the default 2.4.22 kernel scheduler, and even showed a performance gain against the experimental 2.6.0-test9 kernel.

5.4.2 The Zinc Userspace Scheduler

The Zinc userspace scheduler or administrative domain scheduler is implemented in userspace and is a multithreaded server process that accepts connections and messages from other schedulers as well as the monitoring daemons in its own administrative domain. A thread is started whenever it receives a message, and the thread ends when it has finished pro-

cessing the message. The Zinc userspace scheduler monitors the grid processes running on its execution domain, and decides when and where to migrate processes. It also negotiates with other administrative domain schedulers on the exchange of processes to migrate amongst administrative domains.

The Zinc userspace scheduler maintains two tables of information about the state of its administrative domain:

1. The state of all grid processes that is currently executing in any of the nodes in the execution domain

2. All execution hosts known to it, whether they are in its own execution domain, or hosts in other administrative domains which has announced themselves to the Zinc userspace scheduler. Both sets of information are maintained in the same table, making it possible to sort all the available execution nodes by ranking the "desirability" of each node. Each host also has meta-information associated with it:

    - The load history of the node, in an array of floating point numbers of type `float`. The last 20 values are recorded.
    - A weight variable of type `double` that is enumerated to assign ranks to the node.

3. A list of other administrative domains it knows about, represented by the IP addresses of the Zinc userspace schedulers of each administrative domain

Every time a new process starts in an execution domain (the execution domain where the process is initiated is called the *home execution domain*), the process launcher on the host node in the execution domain will send the following information to the Zinc

userspace scheduler governing that particular administrative domain (which is called the *home administrative domain*):

1. The process id of the newly started program
2. The user that started the program
3. The hostname and IP address of the node that started the program
4. The program name
5. The time which the program began
6. Current working directory of the program

Additionally, the following information about the processes is maintained by the Zinc userspace scheduler:

1. IP address of the host that the running process was currently started or restarted on.
2. The number of times the process has been checkpointed
3. The source administrative domain that the process originated from
4. The program execution state, whether it is in the CPU run-queue or checkpointed
5. The program's unique ID, which is an integer that uniquely identifies any grid process within a particular administrative domain.

If the process is a newly started process (as opposed to a restarted checkpoint), then a unique id is assigned to it. The unique id is an integer of type `long` (which is 32 bits on

i386). The unique id starts at 0, and increments for each process that gets started within a particular execution domain. Once the process is running, the monitoring daemon on the node that the program started executing in will give the Zinc userspace scheduler additional information about the process:

1. "Nice" value (priority).
2. RSS or resident set size of the process (the amount of physical memory the process is currently using, minus pages that have been swapped out).
3. Virtual memory footprint of the process (the total amount of memory the process takes up, including the pages currently not in physical memory).
4. Current CPU usage of the process (the number of jiffies the process has been scheduled in either kernel mode or user mode on this current host for this host since it was executed or restarted)

When a process is checkpointed, the following information is embedded together in the checkpoint file:

1. Original working directory.
2. The names and contents of the files which the process opened.
3. The unique ID assigned by the home administrative domain Zinc userspace scheduler when the process was first created.
4. The IP address of the original host it was started on in its home execution domain.
5. The number of times checkpointed
6. The original priority assigned by the home administrative domain

7. The IP of its home administrative domain scheduler node

8. Program name

The checkpoint file is then ready for migration, and is sent to the node within its administrative domain on which the Zinc userspace scheduler resides. When a target system is found by the Zinc userspace scheduler, the entire file is copied over via ssh to the target host. If the target host is in another administrative domain, a message requesting a process transfer is made to the Zinc userspace scheduler that governs the foreign administrative domain. The message contains the following information:

1. The origin administrative domain

2. The requested node in the target execution domain

3. Process id and virtual memory size of the process

The target administrative domain userspace scheduler will evaluate whether there is sufficient memory on the target node, and also whether the memory on that node is sufficient to accommodate the virtual memory size of the process. A message is sent in reply to the requesting administrative domain userspace scheduler either accepting or rejecting the process transfer request (figure 5.6).

When an administrative domain agrees to receive a checkpoint file, it is received, sent to the target host in the execution domain, unpacked into a new temporary working directory with all its files on the given node in the execution domain, and finally the Zinc userspace scheduler restores embedded values of the unique ID, home administrative domain userspace scheduler IP and the number of times the process has been checkpointed in the process table. The program name does not really mean anything to the remote

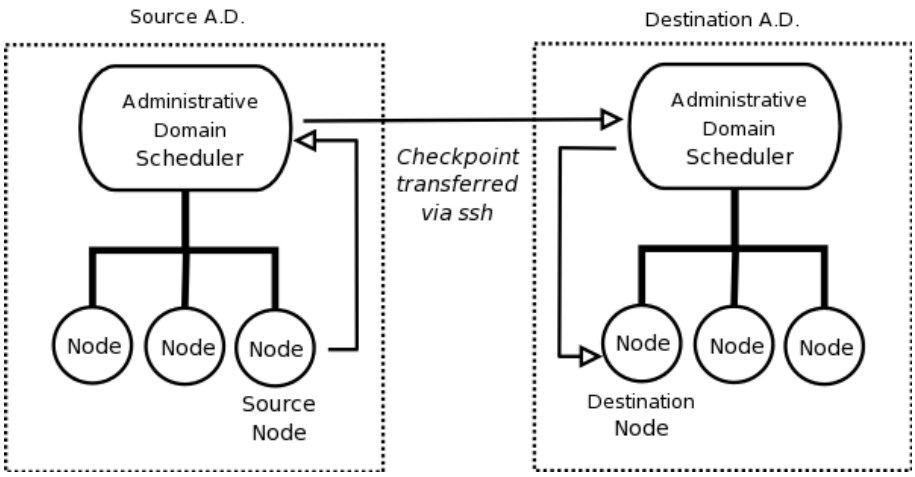

Figure 5.6: Checkpointing And Migration

Zinc userspace scheduler, so it is not restored, and the checkpoint executable name is used instead. The priority of the process will run at a "nice" value one greater than the highest "nice" value for the set of grid processes that belong to the home execution domain (hence at a lower priority). However, if the "nice" priority is at the maximum value of 20, then it is maintained at 20.

There is one thread in the Zinc userspace scheduler that checks the state of the administrative domain at a fixed interval of roughly 20 seconds[2]. Whenever it needs to inspect the state of the system and make scheduling decisions based on the assessment, the Zinc userspace scheduler is said to have reached a *scheduling point*. At each scheduling point, the Zinc userspace scheduler will query the monitoring daemons on every node in the execution domain. The monitoring daemons will then return information about every grid process that is running on their respective nodes, and information about the host itself. The information that is updated for the processes is priority, RSS, virtual memory usage and CPU usage. After the information is updated, the scheduler will make checkpoint and migration decisions.

[2]Since the version of GNU/Linux used does not support hard realtime, there is no guarantee that exactly 20 seconds elapse between intervals, and thus the interval is somewhere between 20-35 seconds

5.5 Resource Discovery And Monitoring

Resource discovery is implemented in the Zinc userspace scheduler. When the Zinc userspace scheduler first starts up, it loads an initial set of addresses of other administrative domains from a configuration file. It is assumed that users participating in the grid know of at least one other administrative domain. The rest of the administrative domains can be dynamically discovered during execution via the Name Dropper algorithm. However, in the current implementation of Zinc, only host information is discovered on the network, it is assumed they are who they say they are; there is no propagation of identification credentials in this implementation as security details are beyond the scope of this research. However, it is possible to implement a public key infrastructure based certifying authority that can verify the authenticity of hosts. Currently, the trust model is commutative, but more sophisticated trust models can also be considered. See section 7.5 for details.

After the Zinc userspace scheduler starts up, the nodes in the execution domain will fire up the monitoring daemons. The first thing these daemons do is report to the Zinc userspace scheduler of their existence, and then each daemon supplies the following information:

1. Its current CPU load.
2. Its current available memory.
3. The percentage of memory usage.

Within the scheduling thread, the Zinc userspace scheduler will broadcast information about all its execution domain nodes to other administrative domains using the Name Dropper algorithm. At every scheduling point, the Zinc userspace scheduler will ask every

node in the execution domain to update the CPU load and memory information by sending a message to them. Then it will select one administrative domain at random from its list of known administrative domains, and send the CPU load and free memory information of each node to the target administrative domain chosen in a network message. Similarly, at certain intervals, the Zinc userspace scheduler for a particular administrative domain will receive information about other execution domain hosts from foreign administrative domains. The Name Dropper algorithm will propagate the information in $O(log^2 N)$ time. This is done continually to ensure that new values keep propagating through the network of administrative domains, and is required for the Name Dropper algorithm to work.

CPU load, total free memory and process information is all obtained via the `/proc` filesystem on each node by the monitoring daemon. The `/proc` filesystem exposes data from the kernel to userspace, allowing statistics to be collected about the system that only the kernel can track accurately.

5.5.1 Obtaining CPU Load

The CPU load is a function of the number of processes that is in the CPU runqueue. The `/proc/loadavg` file returns the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1, 5, and 15 minutes. The monitoring daemon uses the most recent value, which is the number of jobs averaged over the past 1 minute.

5.5.2 Obtaining Total Free Memory And Usage

The total free memory of a particular node can be found by parsing the `/proc/meminfo` file. The kernel doesn't really keep a lot of memory free and idle. Most of the free RAM is used for the caching of frequently used pages, but this cache is released to become free memory when the user needs more RAM. Therefore, when calculating the free available

memory of a particular node, it totals the following:

1. Free available memory
2. Memory used for cached pages
3. Memory used for miscellaneous buffers

The total value is what the Zinc system defines as "free memory":

$$mem_{freetotal} = mem_{free} + mem_{cached} + mem_{buffers} \tag{5.1}$$

The memory usage ratio is defined as:

$$mem_{ratio} = \frac{mem_{total} - mem_{freetotal}}{mem_{total}} \tag{5.2}$$

5.5.3 Preventing Overloading

The monitoring daemon enforces a grid process limit to prevent the node from being swamped with processes. There are two ways it accomplishes this. When a node exceeds its load or memory usage threshold, it will do the following:

1. Rejects migration requests to the node.
2. Tries to migrate out grid processes to other nodes.

When the Zinc userspace scheduler detects the load or memory usage threshold are past the limit, it will initiate the process selection mechanism for checkpointing and migration to another host. However, the Zinc daemons have no authority to checkpoint or migrate

non-grid process (i.e. processes not started with the special `gstart` program which is used to launch grid processes), hence the node becomes almost unusable if it becomes overloaded with non-grid processes. In this event, the scheduler will reject any migration requests to that particular node, and will immediately migrate out any newly started grid processes at the next scheduling point.

When evaluating our system, we needed to compare preemptive migration with non-preemptive migration (initial placement). Therefore, we created a version of the Zinc scheduler which disables checkpointing and migration and uses the Zinctask algorithm to perform initial placement instead. When checkpointing and migration is disabled and the node is overloaded, no new grid processes can be created there and any attempt to do so is denied by the monitoring daemon and a message is sent to the Zinc userspace scheduler informing it of the grid process creation failure.

5.6 Command Line Tools

Zinc includes several command line tools that is used to submit processes to the system, as well as manual checkpointing utilities. The `gstart` program is used to start "grid processes" on any node. The `gstart` program requires the Zinc userspace scheduler and monitoring daemons to be active before it can work. It takes two arguments; the first is the program name that the user wants to start. The second argument is optional, and represents the "nice" value the program will be started at. If none is specified, then the default "nice" value of 0 is assumed.

The second command line tool is the `ckpt` program that can manually checkpoint running processes. It accepts the process id of the desired process to be checkpointed as its only parameter and creates a default checkpoint image called `checkpointed`. This file

can be renamed, and to resume the program one needs only to execute it like any other executable binary. This tool is useful for debugging.

For future work, it would be possible to create a specialised "grid shell" that users can log in to and any programs they start will automatically become "grid processes".

5.7 Summary

Our implementation of Zinc relies heavily on existing checkpoint code written by Eduardo Pinheiro. We have cleaned up and ported the implementation to a more recent version of the 2.4 Linux kernel (2.4.22), and fixed support for opening and closing and reopening of files. The process checkpointing API provided by the EPCKPT patch gives Zinc the ability to create process images that can be migrated and restarted. The Linux kernel scheduler that we used in Zinc was developed specifically to cater to grid processes by giving slighter longer CPU time quantums, thus providing a benefit for long-running CPU intensive processes. Apart from the local kernel scheduler, there exists an administrative domain scheduler in userspace (the Zinc userspace scheduler) which is implemented in C as a multithreaded daemon. This userspace administrative domain scheduler handles the actual placement of processes in the grid by checkpointing and migrating processes. On each node, a monitoring daemon reports state information of the node to the Zinc userspace scheduler. The monitoring daemon also prevents the creation of grid processes on overloaded nodes when checkpointing and migration is disabled. The `gstart` command is used to start grid processes.

CHAPTER 6

EXPERIMENTS AND DISCUSSION

6.1 Introduction

In this chapter, we present an experimental evaluation of the Zinc's performance and process management capabilities. Section 6.2 gives details about our experimental environment which document the hardware and software we used. We have split the experiments into two parts. The first part in section 6.3 details the testing of process migration on a local area network and also over the internet. The second part in section 6.4 deals with an investigation into the factors that influence job throughput on our grid test-bed. Section 6.5 is the discussion of the results followed by a summary in section 6.6.

6.2 Experimental Environment

To evaluate wide-area process migration in Zinc, we set up four different administrative domains, three inside the School of Computer Science in Universiti Sains Malaysia, Penang, Malaysia and the fourth in a personal residence in Long Island, New York, United States (see figure 6.1). The administrative domains are named after the hostname of the machine that runs the Zinc userspace scheduler. The hardware specifications to administrative domains 1, 2, 3 and 4 are listed in tables 6.1, 6.2, 6.3 and 6.4 respectively.

Function	**Hostname**	**Num**	**CPU type**	**Memory**	**Network Adapter**
Scheduler	frodo	1	1.70 GHz P3	256 MB	3Com 59x (eth0)
Node	samwise	1	1.70 GHz P3	256 MB	3Com 59x (eth0)

Table 6.1: Administrative Domain 1 (Frodo) in USM, Penang

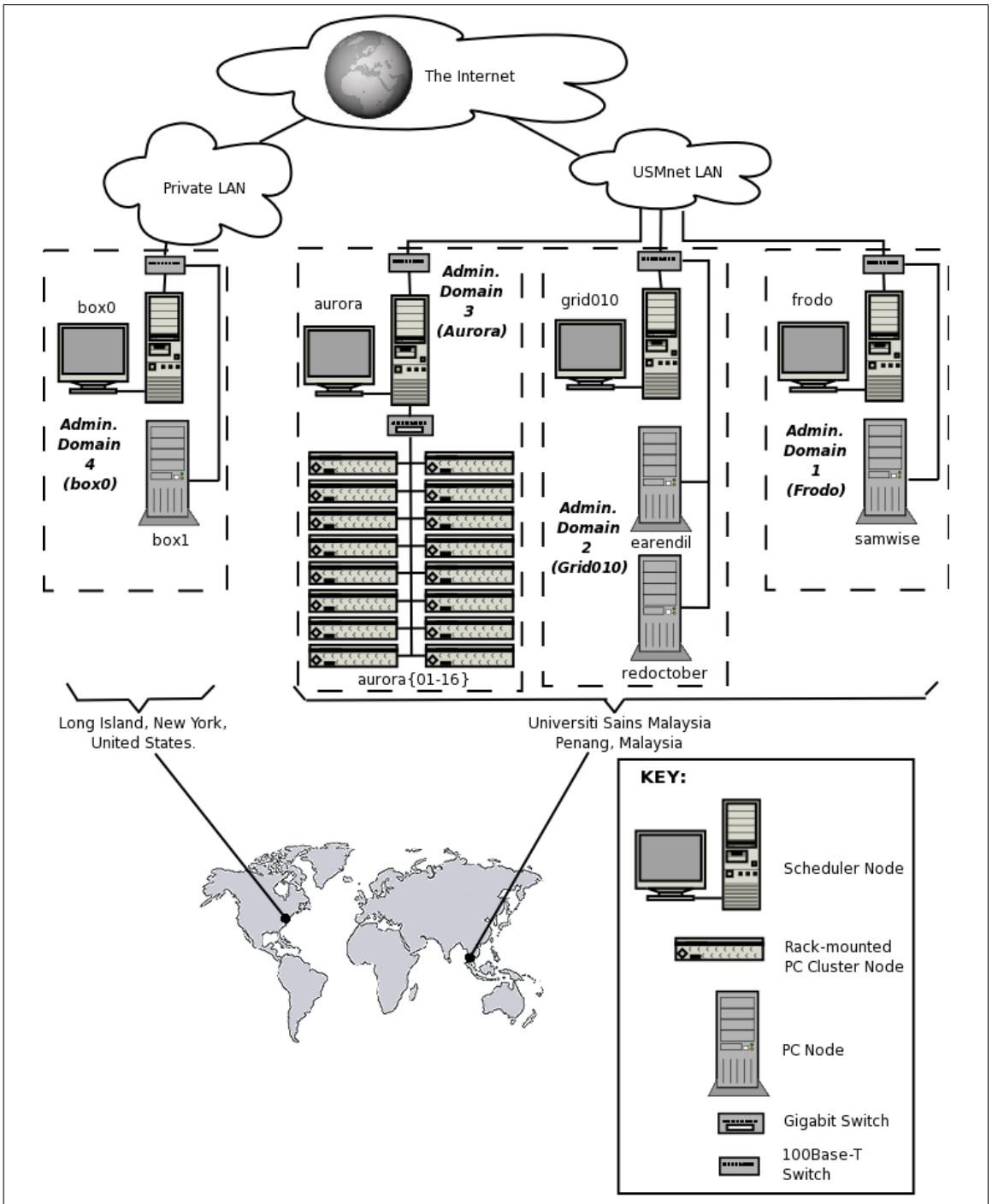

Figure 6.1: Layout Of Hardware In The Zinc Grid Test-bed

Function	Hostname	Num	CPU type	Memory	Network Adapter
Scheduler	grid010	1	1.70 GHz P4	256 MB	3Com 59x (eth0)
Node	earendil	1	1.70 GHz P4	256 MB	3Com 59x (eth0)
Node	redoctober	1	1 GHz P3	256 MB	3Com 59x (eth0)

Table 6.2: Administrative Domain 2 (Grid010) in USM, Penang

Function	Hostname	Num	CPU type	Memory	Network Adapter
Scheduler	aurora	1	1.40 GHz P3 dual cpu (SMP)	2 GB	Intel 82545EM GigE (eth0) Intel 82557 (eth1)
Node	aurora{1–16}	16	1.40 Ghz P3 dual cpu (SMP)	1 GB	Intel 82545EM GigE (eth0)

Table 6.3: Administrative Domain 3 (Aurora) in USM, Penang

Function	Hostname	Num	CPU type	Memory	Network Adapter
Scheduler	box0	1	233 MHz P2	64 MB	3Com 59x (eth0)
Nodes	box1	1	266 MHz P2	64 MB	3Com 59x (eth0)

Table 6.4: Administrative Domain 4 (Box0) in Long Island, New York

Administrative domains 1, 2 and 3 are connected together on a 100BaseT Fast Ethernet switched network, and thus have ample bandwidth to communicate and transfer checkpoint images with one another (Administrative Domain 3 uses a gigabit switch to communicate within its domain, the connection to domains 1 and 2 is limited to 100BaseT). Administrative domain 4 however is connected to the rest of the administrative domains via the internet, and thus has higher lag times. An average connection speed of around 500 Kbps is available between administrative domain 4 and the rest of the administrative domains in Malaysia, though the actual attained speed at any particular moment depends on the amount of traffic and congestion on the network at that time.

6.3 Wide Area Process Migration

6.3.1 Experimental Design And Methodology

The purpose of this part of the experiment is to determine the feasibility of wide-area process migration by determining the time taken for a small and average-sized process to migrate both on a LAN and across the internet. Administrative domains 1, 2 and 4 were used in the setup of the wide area process migration test. The domains were chosen to represent and test specific instances of LAN migration and internet migration

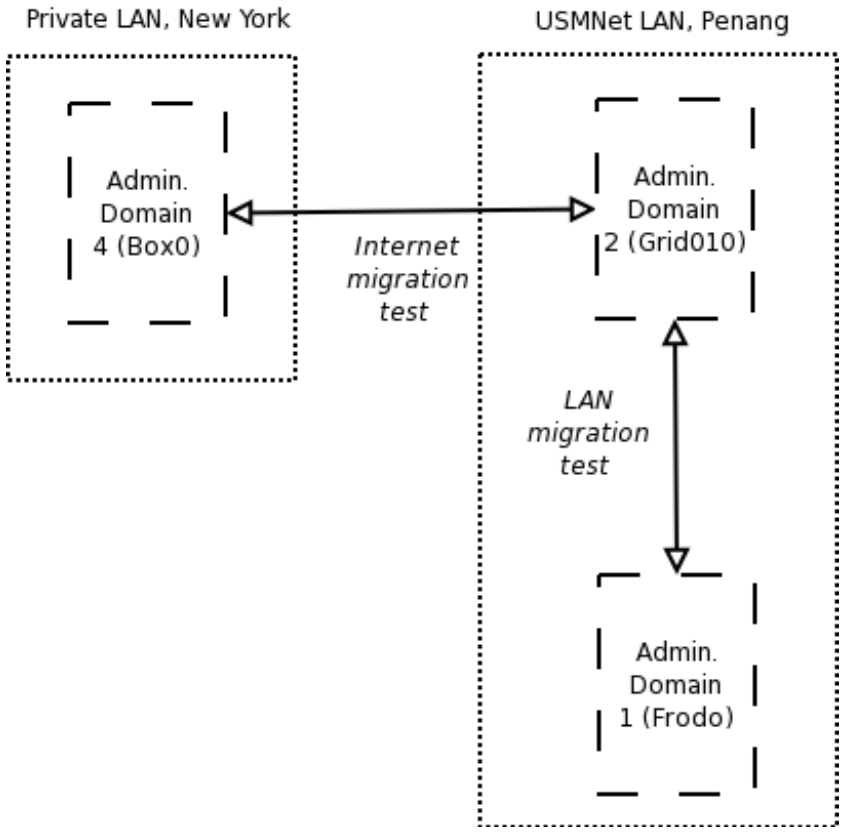

Figure 6.2: Migration Test Configuration

(see figure 6.2). Therefore, the test was divided into two sub-tests, local-area migration (migration over a LAN) and wide-area migration (migration over the Internet). We used two programs, `p1` and `p2`, that had fixed virtual memory sizes of approximately 5 megabytes and 25 megabytes respectively. The resulting checkpoint sizes are 5,335,040 bytes for `p1` and 26,306,560 bytes for `p2`. These virtual memory sizes are typical of CPU-intensive processes that we observed running on the machines in the grid test-bed. We assume that large amounts of data files do not need to be transferred for these kinds of processes.

The processes were migrated between administrative domains 1 and 2 for the local area migration test and between administrative domains 4 and 2 for the wide area migration test. Migration time is the time from the point a process gets checkpointed on the source host to the time it is restarted on the destination host. This includes the time taken to checkpoint the program, transfer the checkpoint image, and restore the checkpoint on the

remote host. Administrative domain 2 is the destination for each migration in all the tests. Each test was carried out 3 times, and the mean of these runs were calculated.

6.3.2 Experimental Results

The mean results of wide area-process migration over 3 runs are shown in tables 6.5 and 6.6.

Program	Run	Time to migrate (seconds)
p1	1	3
p1	2	3
p1	3	3
p1	Mean	3.0
p2	1	11
p2	2	10
p2	3	10
p2	Mean	10.3

Table 6.5: Local-area migration times for processes `p1` (5,335,040 bytes) and `p2` (26,306,560 bytes)

Program	Run	Time to migrate (seconds)
p1	1	85
p1	2	89
p1	3	95
p1	Mean	89.7
p2	1	455
p2	2	387
p2	3	414
p2	Mean	418.7

Table 6.6: Wide-area migration times for processes `p1` (5,335,040 bytes) and `p2` (26,306,560 bytes)

It is clear from the results above that migration only takes several seconds for local migration, and thus the overhead of migration for processes even with relatively large memory footprints (totalling several megabytes) is within reasonable bounds, given sufficient bandwidth. Internet-based migration takes a longer time to accomplish, though the worst case for the migration of `p2` is still only at 7 minutes 35 seconds. The grid was designed to

accommodate jobs that will run for hours or even days, and an overhead of several minutes is negligible. Also, dedicated research networks offer far greater internet bandwidth than our grid test-bed, thus reducing the potential migration time even more for a real system. Thus, wide-area grid process migration is a feasible mechanism to implement as far as atomic processes are concerned. When even more bandwidth is available, it is possible to get better data transfer rates. Barak *et al.* [4] achieved an average process migration rate of 102.6 MB/s with their system, due to the availability of better networking facilities (gigabit 1GB/s Ethernet within their campus network and Ethernet 100MB/s connection across towns that are 10 kilometres apart).

6.4 Factors That Influence Job Throughput On The Grid : Experiments And Results

6.4.1 Experimental Design And Methodology

For the second part of the experiment, we wish to find out the impact of process migration, the Zinctask algorithm, administrative domain size and power and the length of the majority of processes in the system on the job throughput of our grid testbed. In investigating the chosen factors that influence job throughput, we chose to do a factorial experiment. Factorial experiments are conducted to find the effect of pre-determined factors on the mean value of a response variable. The response variable is the metric we want to evaluate the effectiveness of our algorithms. The response variable is *yield*, which is the number of processes completed at any single administrative domain in one experiment run. The value of *yield* represents the throughput achieved by an administrative domain participating in the grid. One of the goals of this research is to evaluate the effectiveness of distributed process management with respect to the factors which we have chosen to study. The factors which we found interesting to look at were the size and computing power of administra-

tive domain, length of the majority of the jobs, preemptive migration vs. initial process placement and the Zinctask host selection algorithm vs. random placement. Therefore, we chose a 4 factorial design for the throughput experiment. In this model, each combination of the factors is tested for interaction.

In our experimental model, we chose to represent the factors by choosing discrete values for each of them:

1. The computing power and size of the administrative domain (Point of entry)
    - Administrative domains 1,2 and 3 were used. Administrative domain 1 (frodo) represents a low compute capacity domain, administrative domain 2 (grid010) represents a mid-size domain and administrative domain 3 (aurora) represents a high-powered, high compute capacity domain. We used the domains that had ample bandwidth and minimum latency amongst each other, therefore those factors are not represented in the model. Therefore, the assumption in our model is ideal network conditions with high bandwidth availability and reasonable amounts of latency (no connection time-outs).
    - The administrative domains represent the points where the jobs enter the system, or point of entry.
    - We could not represent a wider range of hardware, which was limited by the equipment we had access to. However, each administrative domain is relatively different from each other in terms of compute capacity. This diversity helps build a model of a grid where many types of computers are pooled together.
2. Length of the majority of jobs (Length of process majority)
    - We did not want to use a homogeneous batch of jobs of the same length (as

such conditions do not occur in real systems), therefore we mixed a variety of jobs of different lengths.

- The different levels for this factor are 3 sets of job batches where the ratio of short processes to medium-length processes to long processes are 70:15:15, 15:70:15 and 15:15:70 (see table 6.7).
- The length of the jobs is relative to each other. On a lightly-loaded Pentium 4 box, "long" jobs take roughly 45 minutes, "medium" jobs take 15 minutes and "short" jobs take 5 minutes. The run-times vary according to cpu type and load conditions.

3. Preemptive migration vs. non-preemptive migration (Checkpointing vs. no checkpointing)

   - Two systems were used, Zinc where checkpointing was available, and a control program where checkpointing is disabled.
   - While there have been previous studies of the effectiveness of process migration, we want to look at how process checkpointing and migration interacts with the other chosen factors.

4. Host selection algorithm - Zinctask vs. random placement (Placement strategy)

   - The Zinctask algorithm is evaluated by comparing it against random placement. Random placement was chosen as a control mechanism.
   - Random placement is a simple approach to process placement which arbitrarily selects hosts without discrimination. If Zinctask was ineffective at host selection, the results would show its performance to be nearly equal to (or much worse than) random selection.

The factors and their associated levels are summarised in Table 6.8.

Label	% Of Short Jobs	% Of Medium Jobs	% Of Long Jobs
Majority = Short	70	15	15
Majority = Medium	15	70	15
Majority = Long	15	15	70

Table 6.7: Ratios of long to medium to short jobs

Factor	Levels
Point of entry	aurora, grid010, frodo
Length of process majority	long, medium, short
Checkpointing	yes, no
Placement strategy	Zinctask, random

Table 6.8: Factors studied in the throughput experiment

In our experiment runs, we have tried to assign as much jobs as possible to the administrative domains, and thus overloading some nodes.

1. During every run, administrative domains 1, 2 and 3 will each be assigned to execute 60 jobs.

2. The jobs arrive once every 60 seconds.

3. The duration of each run is 1 hour.

4. Every combination of the factors is used to generate runs. There are 4 factors, 2 of them have 3 levels and another 2 have 2 levels. Each combination is replicated in 2 runs[1]. Therefore, a total of $3 \times 3 \times 2 \times 2 \times 2 = 72$ runs are needed for every combination[2].

5. The order of which each run is performed is completely randomised. This is done by shuffling an enumerated list of runs needed to be executed, and then doing each run following the randomised list.

[1] A minimum of 2 replicates is necessary to determine the sum of squares due to error if all possible factorial interactions are considered in the model

[2] However, since all the administrative domains need to be used simultaneously, only 24 runs were performed as each administrative domain will each yield a throughput result on each run (hence 3 results for each domain)

Another aspect of the experimental design which needs to be noted is the metric of throughput. We are not concerned with the total system throughput (i.e., the total number of jobs completed by the entire grid), but instead we look at how many jobs each administrative domain completes for its users while participating in the grid with other administrative domains. For example, if Alice is a user of the Aurora domain, we want to see how many jobs completed that Alice submitted to the grid via the administrative domain she belongs to as a user. At the same time, Bob, who is a user in the Grid010 domain, will also submit his jobs to the grid. Each user's processes will be distributed to any available machine connected to the grid (depending on the load distribution algorithm used), and every machine in the grid is shared amongst all users. At the end of an hour, Alice may have had x number of jobs completed, and Bob may have had y number of jobs completed. Therefore, the throughput for the Aurora administrative domain is x processes an hour and the throughout for the Grid010 domain is y processes an hour, when both are simultaneously accepting jobs and sharing each other's resources.

We believe this is a more interesting metric than simply how many jobs the entire system completes as a whole[3], because it gives us an idea of whether or not there is a benefit for users who belong to particular administrative domain when their resources are pooled together with other administrative domains in a grid.

6.4.2 Experimental Results

The results of the runs are tabulated in table 6.9. The scatter-plot of residuals (see figure A.1 in appendix A) show the results obtained are of a normal distribution [4]. From the univariate analysis of variance, it was found that the following combinations of factorial in-

[3] If we wanted to determine the total system throughput in the Alice and Bob scenario, it would be $x + y$ processes completed an hour

[4] A normal distribution is required for our factorial experiment model to work

Process Length	Admin. Domain	Checkpointing		No Checkpointing	
		Random	ZincTask	Random	ZincTask
Majority long processes	Aurora	26	25	35	29
		25	27	33	22
	Frodo	21	20	0	18
		18	24	1	27
	Grid010	8	11	3	15
		7	13	3	7
Majority medium processes	Aurora	38	38	33	44
		45	41	39	37
	Frodo	32	32	0	36
		31	40	0	36
	Grid010	17	7	2	27
		11	21	1	27
Majority short processes	Aurora	48	53	54	50
		53	52	54	47
	Frodo	44	52	7	48
		52	52	14	52
	Grid010	34	46	34	48
		28	39	16	51

Table 6.9: Process throughput experiment data (each cell contains number of processes completed per hour for each factorial combination with 2 replications)

teractions are statistically significant (we used univariate analysis of variance for statistical significance tests, see appendix A for full ANOVA table):

1. Checkpointing × Placement strategy × Point of entry
2. Length of process majority × Point of entry
3. Length of process majority × Checkpointing × Placement strategy

The following are profile plots of all the significant interactions.

6.5 The Factors That Influence Throughput : Discussion

The profile plots for the significant factorial interactions reveal interesting patterns of the effect of process migration on throughput. They are described in the following subsections.

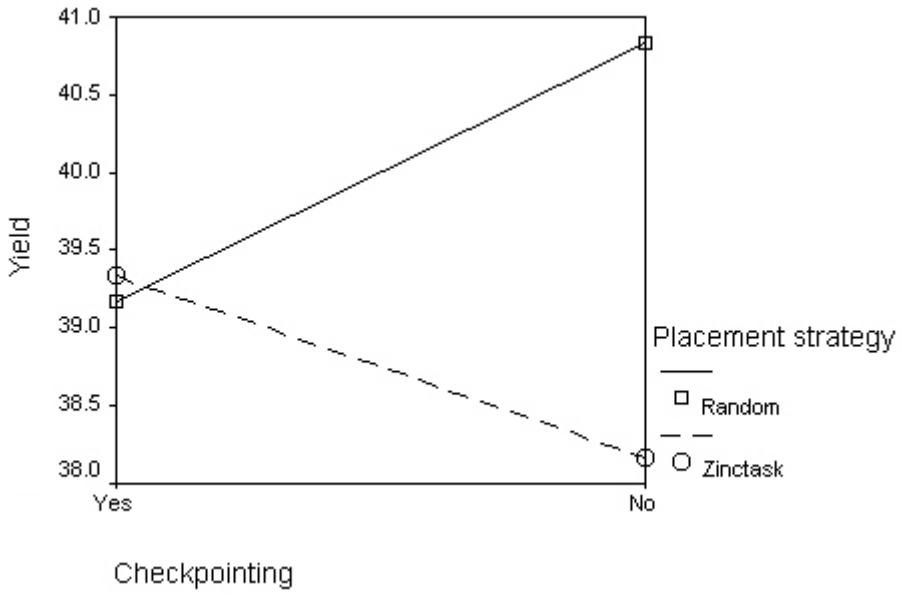

Figure 6.3: Checkpointing × Placement Strategy × Point of Entry Interaction At Aurora

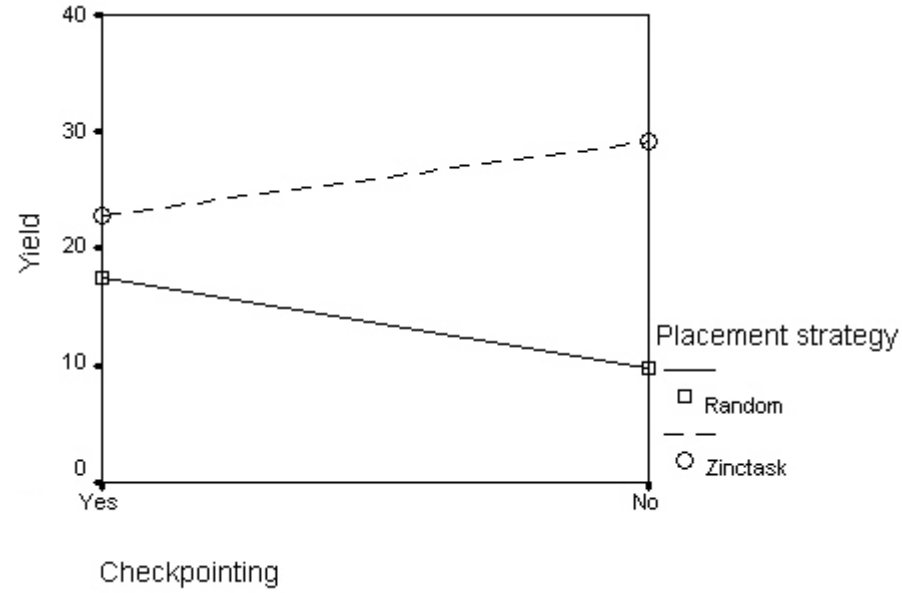

Figure 6.4: Checkpointing × Placement Strategy × Point of Entry Interaction At Grid010

6.5.1 The Effects Of Checkpointing And Preemptive Migration

In the first set of factorial interactions (Checkpointing × Placement strategy × Point of entry), figures 6.3, 6.4 and 6.5 indicate that preemptive migration is beneficial for the job throughput capacity of low-powered administrative domains. By helping its jobs find suitable execution spots in foreign domains, the low-powered administrative domains could

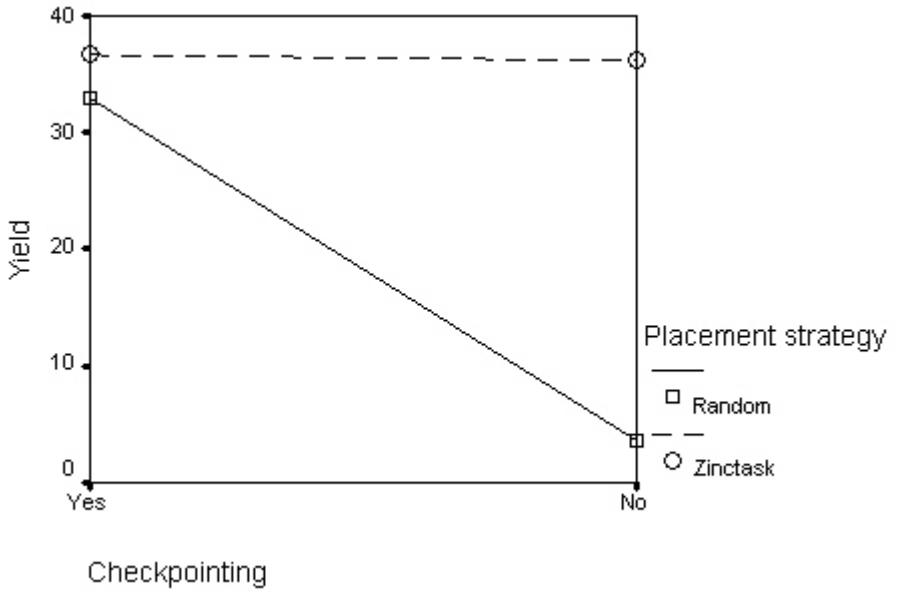

Figure 6.5: Checkpointing × Placement Strategy × Point of Entry Interaction At Frodo

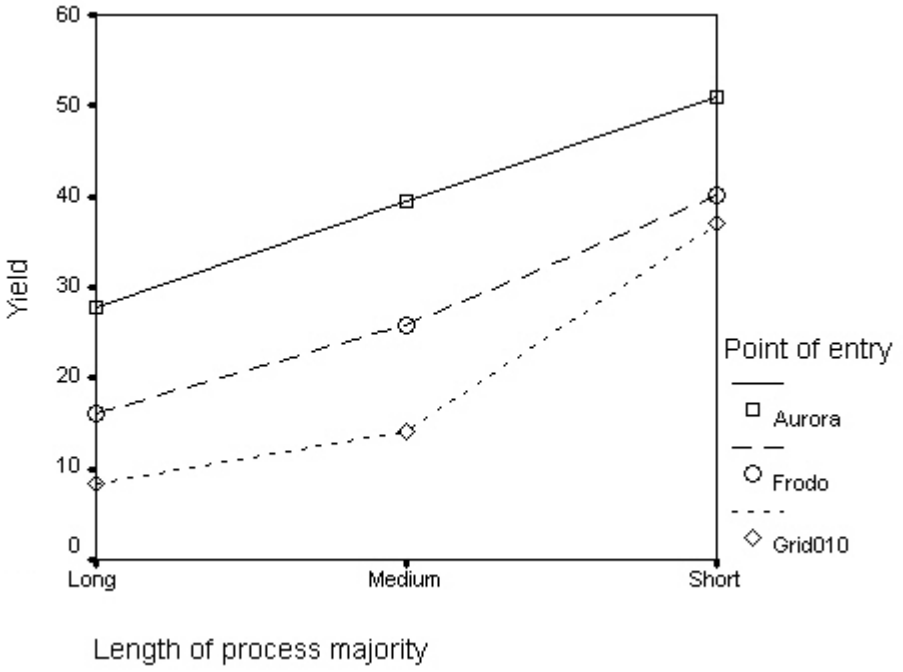

Figure 6.6: Length of Process Majority × Point of Entry Interaction

execute all the jobs we assigned despite the fact that they overwhelm the computing capacity of some nodes. The Frodo administrative domain was the most underpowered, thus found most of its processes shipped to the other more powerful ones such as Aurora. This led to the Frodo domain getting even higher throughput than the Grid010 domain despite

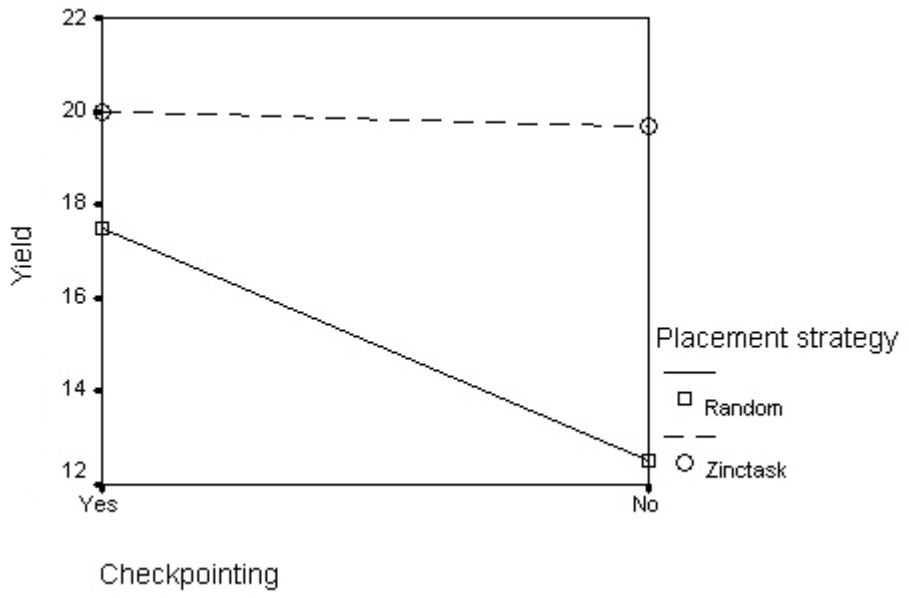

Figure 6.7: Length of Process Majority × Checkpointing × Placement Strategy for Majority Long Processes

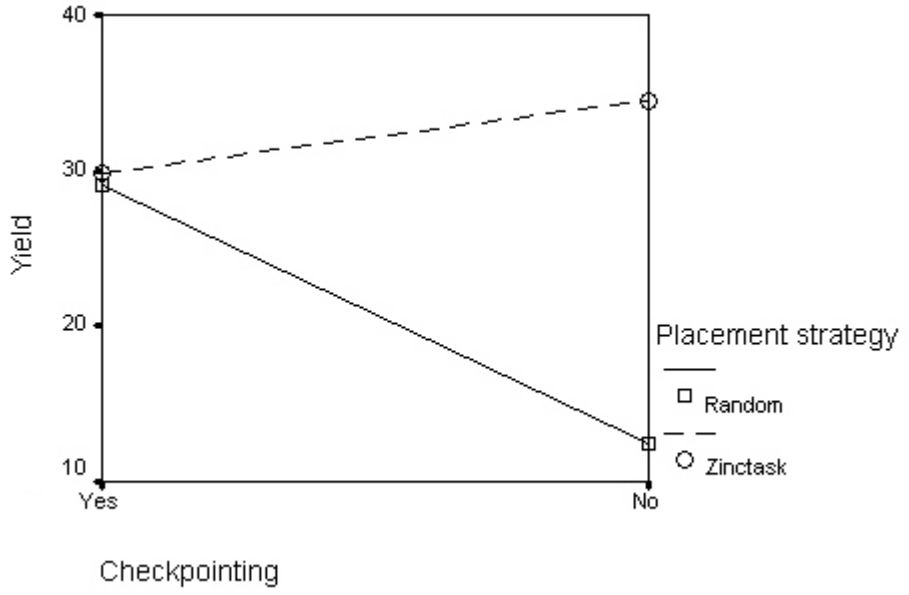

Figure 6.8: Length of Process Majority × Checkpointing × Placement Strategy for Majority Medium Processes

being less powerful. Preemptive migration particularly helps when placement is random, since random placement sometimes leads to bad decisions (as assigning a job to a busy host when idles ones are available). Moving jobs preemptively helps negate those bad decisions. For higher-powered administrative domains, preemptive migration does not seem to affect

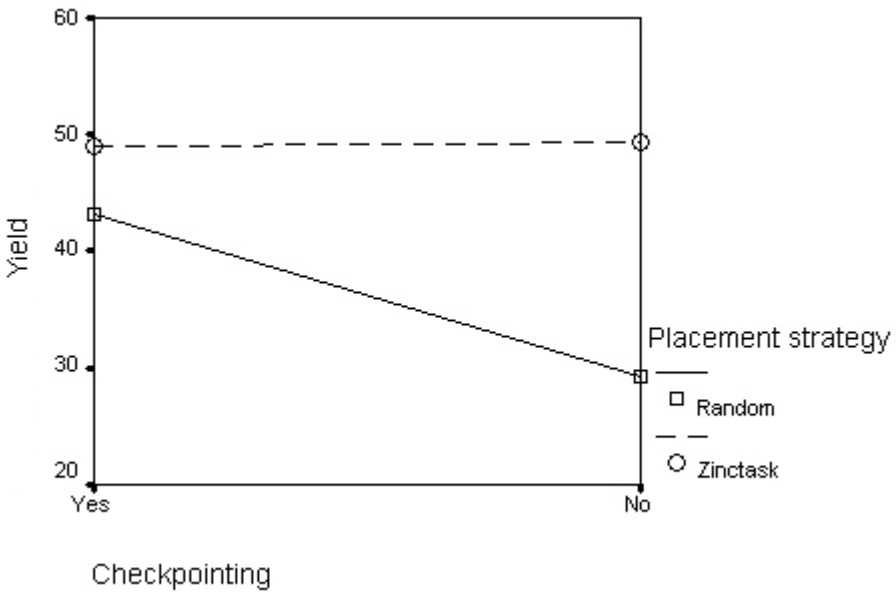

Figure 6.9: Length of Process Majority × Checkpointing × Placement Strategy for Majority Short Processes

throughput much. This could be due to sufficient processing power available for the high-powered domains that migration does not provide a massive improvement in throughput in relative comparison to underpowered administrative domains that get a throughput boost from being able to share the resources of more powerful computers in other administrative domains. There is a slight drop in throughput for the Grid010 domain when the Zinctask algorithm is used together with checkpointing, and an improvement when random placement is used. There seems to be a a very pronounced difference in throughput for the Aurora domain in figure 6.3, but the actual divergence isn't as great as the graph suggests (the mean yield difference is less than 3 jobs).

In the third set of factorial interactions (Length of process majority × Checkpointing × Placement strategy), we can further see that checkpointing and migration helps increase throughput for all types of jobs when placement is random. Preemptive migration makes little difference on the type of processes as it performs nearly the same when Zinctask is used. When random placement is used, preemptive migration also helps increase through-

put when there are more longer running jobs than shorter ones (see figures 6.7, 6.8 and 6.9).

6.5.2 The Effects Of The Zinctask Algorithm

The Zinctask algorithm effectively chooses good hosts for running jobs, as the yield for each profile plot for Zinctask is consistently higher than random placement for almost all scenarios, except for the case in figure 6.3 where Zinctask performs slightly less than (though almost on par with) random placement when no preemptive migration is available. The Zinctask algorithm also performs better than random placement for processes of every length, although there is a smaller difference between the two algorithms when preemptive migration is enabled (see figures 6.7, 6.8 and 6.9).

6.5.3 The Effects Of Process Length

From figure 6.6, we can see that job batches with shorter processes are completed more than the batches with longer ones. This is consistent with logical assumption that more shorter jobs can be finished than longer ones in fixed time frame when there is a finite amount of CPU cycles that can be allotted to each one. However, in our experiment we used a simple model to process length, thus our conclusions may not be representative of the bigger picture of how process length distributions correlate with process migration efficiency and the throughput of the system.

6.5.4 The Effects Of Host Administrative Domain Size And Power

As expected, the largest and most powerful administrative domain (Aurora) achieved the most throughput for jobs originating from itself. Even when placement is random, the Aurora domain performs well. This is due to the amount of processing power available

in the administrative domain. In the lesser administrative domains (Frodo and Grid010), throughput drops significanty when there is no checkpointing and random placement. However, the use of Zinctask and preemptive process migration brings the level of throughput up of the other administrative domains, indicating that the available CPU power of Aurora is shared with the other domains.

6.5.5 Weaknesses Of The Experimental Model

There are several limitations to our experimental model, mainly due to limitations rising from the feasibility of the experiments that need to be conducted. The Aurora cluster that we needed for the experiments is being heavily used as a production system. Therefore, we had limited time and opportunity to utilise it as exclusive computer time needed to be booked, during that time all the current grid users could not use it. Therefore, we designed the experiment to be as simple as possible to execute in a limited time frame to minimise the time needed for the machines to be disconnected from other users.

The main limitation in our experiment is the model used to represent the grid. Although the results derived are valid for our grid test-bed, they may not hold true for other configurations. Also, network latency and traffic were not included in the model to keep it simple. To get a better representation of real grids, it would be better to test the algorithms proposed on a wider range of hardware. In the future, perhaps Zinc could be deployed on real grids with many users and different types of equipment to corroborate the results produced in this work.

We have also chosen a fixed effects model to represent levels of process length and administrative domain capacity. Thus, our results are valid for the configuration we used in the experiment, but we cannot draw any general conclusions from it.

However, despite these limitations, the experiment does shed some light on how process migration behaves on a sample grid implementation.

6.6 Summary

The evaluation of Zinc consists of two parts; one tested the efficiency of wide-area process migration and the other tested the interaction between a combination of factors that influence process throughput. These factors are the size and power of host administrative domain, length of the majority of processes submitted, the availability of checkpointing, and the placement strategy of processes. The placement strategy consists of comparing our Zinctask algorithm with a random selection algorithm. From our experiments, the following observations were obtained:

1. The migration time for processes are reasonably small, even for internet migration. Even on a consumer-grade residential DSL connection, the computers in New York could migrate processes with moderate-sized checkpoints (around 25 megabytes) to a host in Malaysia in an average time of 418.7 seconds (around 7 minutes). Long-running processes meant for grid computing tend to run for hours or even days, making migration time negligible as long as sufficient bandwidth is available.

2. Process checkpointing and migration improve throughput when there lacks a good placement strategy. This improvement is greater for longer processes over shorter ones.

3. The Zinctask algorithm helps improve throughput compared to random placement.

4. When Zinctask is used, throughput is not affected much by process checkpointing.

5. Resource-rich administrative domains does not seem to benefit much with respect

to throughput from either process migration or the Zinctask algorithm. In contrast, the throughput improvements are more pronounced for administrative domains with fewer computing resources.

With Zinc, we achieved most of our design goals satisfactorily. The Zinc schedulers successfully manage and distribute processes in a decentralised fashion. The Zinc-based Linux kernel provides sufficient transparency for single process migration. However, the issue of handling IPC among distributed processes and eliminating the remaining residual dependencies is a topic for future work.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Introduction

In this thesis, we have explored the potential of creating a grid operating system. A grid operating system incorporates many of the features of distributed operating systems, but does not assume full control over all the resources. The assumptions that grid middleware need to work with such as time-limited availability of remote resources are supported by features built into the underlying system. In our research, our focus has been distributed process management for grid processes. For our experiments, we created a grid test-bed consisting of 4 administrative domains, each with a different number of nodes and compute power.

In this chapter, we recap our objectives in section 7.2 as well as evaluate our design goals for Zinc in section 7.3. We will revisit the contributions of this thesis in section 7.4 and future work is discussed in section 7.5. A final summary is provided in section 7.6.

7.2 Objectives Revisited

7.2.1 Integrating Process Checkpointing And Grid Process Scheduling Into The Linux Kernel

We sucessfully ported and integrated the the EPCKPT [61] patch to implement process checkpointing in version 2.4.22 of the Linux kernel. We also developed a kernelspace scheduler with the Linux kernel developers [32] to provide local process scheduling that is

optimised to handle grid processes.

7.2.2 Creating A Prototype Grid Process Management System

Previous work on distributed systems have enabled us to implement many of the technologies needed to create Zinc, namely resource discovery, process checkpointing and process migration (see [51] for a comprehensive survey on process checkpointing and migration). We adopted the Name Dropper resource discovery algorithm [29] for our grid information system and we also developed an algorithm for process placement called Zinctask, which extends upon placement algorithms based on run queue [19] by adding consideration for memory usage, bandwidth, latency and information staleness in our host selection algorithm. All these algorithms have allowed us to successfully implement Zinc, and meet our design goals satisfactorily.

7.2.3 Enabling Wide-Area Process Migration

We measured the time taken for processes to migrate processes over the internet, and found that it was adequately fast even under less-than-optimal bandwidth conditions. A 25 megabyte checkpoint was transferred from Penang, Malaysia to New York, USA and restored in under 8 minutes. This indicates wide-area process migration is feasible (as the migration times are short compared to the typical run-times of CPU-intensive grid processes) as long as the processes do not require IPC.

7.2.4 Investigating The Factors Influencing Throughput In The Grid OS

In our investigations into the effect of process migration on job throughput on our test-bed grid, we discovered the following:

1. Preemptive process migration improves throughput compared to non-preemptive migration when initial placement of processes are random.

2. The Zinctask algorithm gives better throughput than random placement in almost all cases, and on par with random placement when it does not.

3. Preemptive process migration does not affect system throughput significantly (for better or worse) when the Zinctask algorithm is used.

4. Both preemptive process migration and the Zinctask algorithm help low-powered administrative domains increase their throughput.

However, we made the assumption that processes are atomic in nature (do not rely on IPC), and our experimental model is a simple fixed-effects full-factorial model with the levels for the factors being discrete values that represent a larger range of values. Therefore, due to the assumptions and simplification of the representative model, these conclusions are only valid for our test-bed, and further research is needed for a more generalised conclusion.

In this thesis, we do not compare Zinc to other similar systems such as Condor or MOSIX, as it would be too coarse-grained a comparison and would not provide meaningful results within the context of our research objectives.

7.3 Evaluation Of Design Goals

7.3.1 Transparency And Residual Dependencies

Transparency and residual dependencies are inter-related; the more residual dependencies that are eliminated or overcome, the more transparent the process migration mechanism becomes. Looking at the implementation of Zinc, we have succeeded in providing a degree of transparency for migrating processes:

1. Processes stop and restart on a foreign node correctly, with the virtual memory area restored completely.

2. No special compilation steps are necessary for a program to be checkpointable, nor is there any requirement for a special programming language or library.

3. All files in use are transferred together with the checkpoint, and will be found by the program when restarting.

With the above, we have provided *location transparency* for processors and a degree of *relocation transparency* for processes. However, there are weaknesses to the process migration facility which Zinc provides:

1. The operating system environment on a migrated host is going to be different from the home node. Thus, if the program accessed any OS-specific or hardware-specific information on its home node (such as the amount of installed RAM), that information will be stale.

2. Inter-process communication will not work across migrated nodes.

3. The system call `getpid()` will return a different PID for a migrated process, since the C library function call does not refer to the grid process id.

4. It is currently not possible for two migrated processes to write to the same file, as distributed filesystem access is not implemented within the scope of this project.

Addressing these weaknesses remain a challenging problem. There is not much that can be done about hardware specifications changing when a process is migrated, since this will be one of the artifacts of process migration (unless a purely homogeneous distributed system is constructed). The PID problem can be addressed with PID reservation, though it raises

a new issue of synchronising reservations throughout the grid. The possible solution to IPC and shared file access is addressed in the next chapter (see section 7.5 on future work).

7.3.2 Throughput

A combination of both preemptive migration and the Zinctask algorithm help increase job throughput rates for the majority of scenarios presented in our experimental model. Both the Zinctask algorithm and preemptive process migration is beneficial to improving the throughput of small or low-powered administrative domains. When Zinctask is used for placement, using preemptive migration neither helps nor hurts throughput significantly in our grid test-bed. Zinctask performs consistently well in the majority of the test cases in this experiment, giving a better throughput yield. When Zinctask is not used and placement is random, we found that preemptive process migration helps improve throughput significantly. This is especially true if the system has more longer-running processes than shorter ones. This finding complements Harchol-Balter and Downey's assertion that process migration helps the performance of longer-running processes [28]. Our design goal of devising algorithms that deliver better throughput is a success on our grid test-bed. However, further work is needed to confirm if these algorithms perform well in other configurations.

7.3.3 Decentralisation

The design of Zinc creates two levels of structure within the distributed system. Inside an administrative domain, the Zinc userspace scheduler is the centre of all grid scheduling activity. All the nodes within an administrative report to only one scheduler, hence the design is centralised. This limits the size of which a particular administrative domain can grow, since the central scheduler presents a bottleneck. For our grid test-bed, the largest

administrative domain consists of a 16-node cluster, which stills performs well with no apparent bottleneck. However, if larger administrative domains need to be constructed, this single-scheduler design may need to be replaced with multiple schedulers balancing the load between themselves.

Across administrative domains, there is no central authority which coordinates activity. Each and every administrative domain is its own independent entity which communicate with every other administrative domain. Since the Name Dropper algorithm requires that a new node only announce itself to one other connected node, it is possible to add administrative domains to the system by simply connecting it to one existing member of the grid. Therefore, it is possible to "grow" the grid in a decentralised fashion by adding administrative domains, and information propagation scales very well due to the efficiency of the resource discovery algorithms.

7.3.4 Adaptability To Resource Pool

The resource discovery algorithm is primarily responsible for adapting the system to varying resources. Since the Name Dropper algorithm works very efficiently, it is possible for each scheduler to keep an almost-current snapshot of the state of the entire system. When new hosts are added to the system, the new information is propagated throughout all the nodes quickly. This information is vital for the scheduler to make decisions for process migration and placement, and to request resources from foreign administrative domains.

7.4 Contributions Revisited

The following are the contributions of our thesis:

1. Adding the **concept of grid process management** to GNU/Linux as this func-

tionality is required for a grid operating system. To achieve this, we needed to have process migration as a feature in Linux. The underpinnings of process migration was facilitated by existing software called EPCKPT which was updated to to be used with a recent Linux kernel.

2. The **Zinc program**, a proof-of-concept implementation of grid operating system facilities comprising a distributed scheduler and a kernelspace scheduler which benefits the throughput of grid processes. These programs utilise the process migration capabilities of the modified Linux kernel and creates a load distribution mechanism for the grid operating system.

3. The **Zinctask algorithm** is used for placement of processes and is designed to maximise throughput. This algorithm was found to be robust performance-wise when compared to random placement.

4. Exploring the **factors which influence process throughput in a grid** and their factorial interactions. The factors are the availability of preemptive processes migration, the placement strategy of processes, the configuration of the machines in the grid and also the length of jobs that are submitted to the grid.

7.5 Future Work

There is much work that needs to be done in producing an actual working grid operating system. In this section, we outline several areas which can be further researched upon in realising a usable grid operating system.

7.5.1 Grid Process Management

In our design of Zinc, we have not included support for inter-process communication, namely sockets, Sys V shared memory and Sys V semaphores. Thus, our process migration mechanism is simplified to only work with "atomic" processes which do not communicate with other processes. The reason for this is a weakness in the implementation, though there are a few recent checkpointing implementations for Linux which do support some IPC such as the migration of sockets [86]. However, the following issues arise when migrating processes which perform IPC:

1. The possibility that residual dependencies will occur between two or more hosts when processes are migrated from a particular machine but need to communicate with other processes on that machine. If migration occurs multiple times, the problem escalates. Residual dependencies can be handled reasonably well if the distance between the hosts are sufficiently short, thus avoiding latency problems. However, there needs to be a method of defining virtual boundaries where processes should not migrate across if they are communicating with processes in a particular machine.

2. A grid-wide namespace is required for all IPC between processes that migrate. Traditionally, TCP sockets use an IP address and a port number, Unix sockets use a filename and Sys V IPC such as semaphores and shared memory use a unique numerical key. If processes were allowed to do IPC across nodes, and even across administrative domains, we would need a global namespace for each of these facilities. TCP sockets already have the benefit of TCP/IP naming, and the others could be implemented via a grid filesystem (see next section).

7.5.2 Grid Filesystems

An active area of research is distributed filesystems, and much of the state-of-the-art can be applied to assist the implementation of a grid filesystem. A grid filesystem would support features like disconnected operations (such as found in AFS and Coda), caching and distribution of large data sets, and opportunistic data movement [78]. However, the most interesting feature that would come from a globally accessible filesystem would be the availability of a global namespace. One of the strengths of Unix-like operating systems is that many things are abstracted as files, and thus the filesystem could be used to implement namespaces for IPC such as Sys V semaphores. Sys V keys could be published globally via the filesystem, and thus enabling distributed processes to perform IPC. It is also possible to share Unix sockets, kernel information (via a distributed `/proc` interface, for example) and other abstractions via the filesystem interface, such as what is done by the Plan 9 operating system.

7.5.3 Security

From a security standpoint, allowing binary code to migrate from another administrative domain to your own and allowing it to run might seem like a frightening prospect if it cannot be trusted entirely. Unix-like systems offer various degrees of control over what a running process can or cannot do (such as Unix permissions and POSIX access control lists), and it is mostly sufficient for most purposes. However, it would be safer for the host operating system to virtualise an execution environment for a guest process, such as via `chroot` or FreeBSD's `jail` [36].

7.6 Summary

Preemptive process migration is a useful facility to have in a grid operating system. Besides the throughput boost when there is a lack of a good placement strategy, it does not adversely affect the system performance as far as we have seen. Therefore, considering the other advantages of having preemptive process migration (administrative domain autonomy, mobile resource location, etc.), it is therefore beneficial to have this facility available.

Grid operating systems provide a fresh look at solving many of the problems in grid computing. If resource allocation and usage are handled transparently at the OS level, it will reduce the application programmers' burden of working with many of the idiosyncrasies of current grid computing middleware. However, grid operating systems are not meant to eliminate the need for grid software such as Globus, only to make it possible to simplify it and complement the functionality provided by grid middleware. The Linux checkpointing patches, the high throughput kernel scheduler, the Zinc scheduling and load distribution software provides a starting point for which a complete grid operating system can be constructed.

From our experience doing this research, we suggest that it is worth supporting grid computing at an operating system level to help implement a simple yet powerful environment for grid systems to run on. Just as traditional operating systems when first introduced helped make computer hardware considerably easier to use, to program for, more coherent and more transparent, we hope that grid operating systems will do the same for grid computing.

REFERENCES

[1] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An Experiment in Public-resource Computing. *Commun. ACM*, 45(11): 56–61, 2002.

[2] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993. ISBN 0387566635.

[3] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4–5): 361–372, 1998.

[4] Amnon Barak, Amnon Shiloh, and Lior Amar. An Organizational Grid of Federated MOSIX Clusters. In *Proc. 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 350–357. IEEE Computer Society, 2005.

[5] Francine Berman. High Performance Schedulers. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[6] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, 2nd edition, 2002.

[7] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[8] Rajkumar Buyya, Jonathan Giddy, and David Abramson. An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications. In *Proceedings of the 2nd International Workshop on Active Middleware Services (AMS 2000)*, Pittsburgh, USA, 2000. Kluwer Academic Press.

[9] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[10] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of USENIX NSDI 2005*, 2005.

[11] Sivarama P. Dandamudi. Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 484–492, Washington, DC, USA, 1995. IEEE Computer Society.

[12] Sivarama P. Dandamudi and K. C. Michael Lo. A Hierarchical Load Sharing Policy for Distributed Systems. In *MASCOTS '97: Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 3–10, Washington, DC, USA, 1997. IEEE Computer Society.

[13] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Softw. Pract. Exper.*, 21(8):757–785, 1991.

[14] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems*, 4(4):353–384, 1991.

[15] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[16] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *SIGMETRICS '88: Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 63–72, New York, NY, USA, 1988. ACM Press.

[17] M. Raşit Eskicioğlu. Design Issues of Process Migration Facilities in Distributed Systems. *IEEE Technical Commitee on Operating Systems Newsletter*, pages 3–13, 1989.

[18] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Technical Report CS-97-05, Department of Computer Science, University of Virginia, March 1997.

[19] Domenico Ferrari and Songnian Zhou. An Empirical Investigation of Load Indices for Load Balancing Applications. In *Performance '87: Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 515–528, 1987.

[20] Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA*, February 2003.

[21] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[22] Ian Foster and Carl Kesselman. Computational Grids. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[23] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid : Enabling Scalable Virtual Organizations. *International Journal Of High Performance Computing Applications*, 15(3):200–222, 2001.

[24] Richard P. Gabriel. Lisp: Good News, Bad News, How to Win Big, 1991. URL http://www.dreamsongs.com/WIB.html. Accessed 30th June 2006.

[25] Andrze Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley Longman Publishing Co., Inc., 1991. ISBN 0201417049.

[26] Andrew S. Grimshaw and William A. Wulf. Legion – A View From 50,000 Feet. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7582-9.

[27] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The Next logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, University of Virginia, August 1994.

[28] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[29] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource Discovery in Distributed Networks. In *PODC '99: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–237. ACM Press, 1999.

[30] Sandra Hedetniemi, Stephen Hedetniemi, and Arthur Liestman. A Survey of Gossiping and Broadcasting in Communications Networks. *Networks*, 18:319–349, 1988.

[31] Philip Homburg, Maarten van Steen, and Andrew S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 75–82. ACM Press, 1996.

[32] Nur Hussein, Constantine Kolivas, Fazilah Haron, and Chan Huah Yong. Extending The Linux Operating System For Grid Computing. In *Proceedings of the APAN Network Research Workshop*, July 2004.

[33] Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, pages 51–62. Springer-Verlag, 2001. ISBN 3-540-42949-2.

[34] Sun Microsystems Inc. *System and Network Administration*, March 1990. Part Number 800-3805-10.

[35] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. FLIP: An Internetwork Protocol for Supporting Distributed Systems. *ACM Trans. Comput. Syst.*, 11(1):73–106, 1993.

[36] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of 2nd International System Administration and Networking Conference (SANE2000)*, Maastricht, The Netherlands, May 2000.

[37] Nirav H. Kapadia, José A. B. Fortes, and Carla E. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 6, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0287-3.

[38] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 213–225, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-447-3.

[39] Phillip Krueger and Miron Livny. The Diverse Objectives of Distributed Scheduling Policies. In *Proc. Seventh Int'l Conf. Distributed Computing Systems*, pages 242–249, Los Alamitos, CA, USA, 1987. IEEE CS Press.

[40] Phillip Krueger and Miron Livny. A Comparison of Preemptive and Non-Preemptive Load Distributing. In *8th International Conference on Distributed Computing Systems*, pages 123–130, San Jose, CA, June 1988.

[41] Krzysztof Kurowski, Jarek Nabrzyski, Ariel Oleksiak, and Jan Węglarz. Multicriteria Aspects of Grid Resource Management. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Węglarz, editors, *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[42] Krzysztof Kurowski, Jarek Nabrzyski, and Juliusz Pukacki. User Preference Driven Multiobjective Resource Management in Grid Environments. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 114, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8.

[43] Byoung-Dai Lee and Jennifer M. Schopf. Run-Time Prediction of Parallel Applications on Shared Environments. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03)*, page 487, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Will Leland and Teunis J. Ott. Load-balancing Heuristics and Process Behavior. In *SIGMETRICS '86/PERFORMANCE '86: Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation*, pages 54–69, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-184-9.

[45] Jin Liang, Klara Nahrstedt, and Yuanyuan Zhou. Adaptive Multi-resource Prediction in Distributed Resource Sharing Environment. In *Proc. 4th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 293–300. IEEE Computer Society, 2004.

[46] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[47] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[48] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), June 1997.

[49] Miron Livny and Rajesh Raman. High-throughput Resource Management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[50] Marek Mika, Grzegorz Waligóra, and Jan Węglarz. A Metaheuristic Approach to Scheduling Workflow Jobs on a Grid. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Węglarz, editors, *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[51] Dejan S. Milojičič, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[52] Dejan S. Milojičič, Peter Giese, and Wolfgang Zint. Experiences with Load Distribution on Top of the Mach Microkernel. In *USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 19–36, 1993.

[53] Andrey Mirtchovski, Rob Simmonds, and Ronald Minnich. Plan 9 – An Integrated Approach to Grid Computing. In *IPDPS 2004 : 18th International Parallel and Distributed Processing Symposium CD-ROM / Abstracts Proceedings*. IEEE Computer Society, April 2004. ISBN 0-7695-2132-0.

[54] Ingo Molnar. [announce] [patch] ultra-scalable o(1) smp and up scheduler. [Online posting]. Linux Kernel Mailing List, 2 January 2002. URL http://lkml.org/lkml/2002/1/3/287.

[55] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[56] Gary Nutt. *Operating Systems: A Modern Perspective*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000.

[57] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23–36, 1988.

[58] Pradeep Padala and Joseph N. Wilson. GridOS: Operating System Services for Grid Architectures. In *Proceedings of the International Conference On High Performance Computing (HiPC'03)*, December 2003.

[59] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS Version 3: Design and Implementation. In *USENIX Summer*, pages 137–152, 1994.

[60] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3): 221–254, Summer 1995.

[61] Eduardo Pinheiro. Truly-Transparent Checkpointing of Parallel Applications, 23 September 2002. URL http://www.research.rutgers.edu/~edpin/epckpt/paper_html/. Accessed 30th June 2006.

[62] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[63] Eric S. Raymond, editor. *The New Hacker's Dictionary*. The MIT Press, 3rd edition, 1996.

[64] Michael Richmond and Michael Hitchens. A New Process Migration Algorithm. *SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[65] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[66] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990.

[67] Jennifer M. Schopf. Ten Actions When Grid Scheduling. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Węglarz, editors, *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[68] Michael Shirts and Vijay Pande. Screen Savers of the World Unite! *Science*, 290, 2000.

[69] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[70] Chris Steketee, Weiping Zhu, and Philip Moseley. Implementation of Process Migration in Amoeba. In *International Conference on Distributed Computing Systems*, pages 194–201, 1994.

[71] Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing for Heterogeneous Architectures. In Dimiter R. Avresky and David R. Kaeli, editors, *Fault-Tolerant Parallel and Distributed Systems*. Kluwer Academic Press, 1998.

[72] Atsuko Takefusa, Satoshi Matsuoka, Henri Casanova, and Francine Berman. A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 406, Washington, DC, USA, 2001. IEEE Computer Society.

[73] Andrew S. Tanenbaum and Sape J. Mullender. An Overview of the Amoeba Distributed Operating System. *SIGOPS Oper. Syst. Rev.*, 15(3):51–64, 1981.

[74] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems – Principles and Paradigms*. Prentice-Hall, Upper Saddle River, New Jersey, 2002.

[75] Andrew S. Tanenbaum and Robbert van Renesse. Distributed Operating Systems. *Computing Survey*, 17(4):419–470, 1985.

[76] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12):46–63, 1990.

[77] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems (2nd ed.): Design and Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-638677-6.

[78] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo Approach to Data Movement on the Grid. In *Proceedings of the Tenth (IEEE) Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, CA, August 2001.

[79] Marvin M. Theimer and Barry Hayes. Heterogeneous Process Migration by Recompilation. In *IEEE 11th International Conference on Distributed Computing Systems*, pages 18–25. IEEE CS Press, 1991.

[80] Mark Wahl, Tim Howes, and Steve Kille. Lightweight Directory Access Protocol (v3). Internet Engineering Task Force RFC2251, December 1997. URL http://www.ietf.org/rfc/rfc2251.txt.

[81] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 22–31, Washington, DC, USA, 1995. IEEE Computer Society.

[82] Yung-Terng Wang and Robert J.T. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, 34(3):204–216, 1984.

[83] Lingyun Yang, Ian Foster, and Jennifer M. Schopf. Homeostatic and Tendency-Based CPU Load Predictions. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003. IEEE Computer Society.

[84] Wengyik Yeong, Tim Howes, and Steve Kille. Lightweight Directory Access Protocol. Internet Engineering Task Force RFC1777, March 1995. URL http://www.ietf.org/rfc/rfc1777.txt.

[85] E. Zayas. Attacking the Process Migration Bottleneck. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[86] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint / Restart as a Kernel Module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

[87] Songnian Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. Softw. Eng.*, 14(9):1327–1341, 1988.

LIST OF PUBLICATIONS

[1] Nur Hussein, Constantine Kolivas, Fazilah Haron, and Chan Huah Yong. Extending The Linux Operating System For Grid Computing. In *Proceedings of the APAN Network Research Workshop*, July 2004.

APPENDICES

APPENDIX A

UNIVARIATE ANOVA OF RESULTS

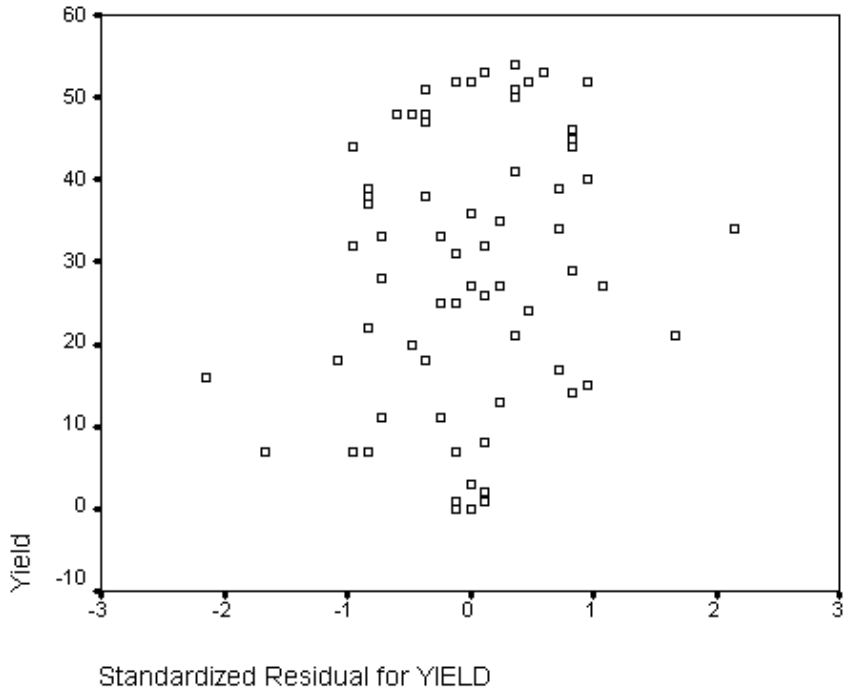

Figure A.1: Scatterplot of Standardised Residuals vs. Yield

		Value Label	**N**
Length of process majority	1	Long	24
	2	Medium	24
	3	Short	24
Checkpointing	1	Yes	36
	2	No	36
Placement strategy	1	Random	36
	2	Zinctask	36
Point of entry	1	Aurora	24
	2	Frodo	24
	3	Grid010	24

Table A.1: Between-Subjects factors

Dependent variable: **Yield**

Source	**Type III Sum of Squares**	**df**	**Mean Square**	**F**	**Sig.**
Corrected Model	19297.611(a)	35	551.360	31.357	.000
Intercept	59973.389	1	59973.389	3410.809	.000
LENGTH	7883.861	2	3941.931	224.186	.000
CHECK	470.222	1	470.222	26.742	.000
STRATEGY	1701.389	1	1701.389	96.761	.000
ENTRY	4662.028	2	2331.014	132.570	.000
LENGTH * CHECK	55.861	2	27.931	1.588	.218
LENGTH * STRATEGY	221.861	2	110.931	6.309	.004
CHECK * STRATEGY	800.000	1	800.000	45.498	.000
LENGTH * CHECK * STRATEGY	205.750	2	102.875	5.851	.006
LENGTH * ENTRY	277.972	4	69.493	3.952	.009
CHECK * ENTRY	867.861	2	433.931	24.679	.000
LENGTH * CHECK * ENTRY	113.306	4	28.326	1.611	.193
STRATEGY * ENTRY	1182.694	2	591.347	33.631	.000
LENGTH * STRATEGY * ENTRY	48.806	4	12.201	.694	.601
CHECK * STRATEGY * ENTRY	753.083	2	376.542	21.415	.000
LENGTH * CHECK * STRATEGY * ENTRY	52.917	4	13.229	.752	.563
Error	633.000	36	17.583		
Total	79904.000	72			
Corrected Total	19930.611	71			
a R Squared = .968 (Adjusted R Squared = .937)					

Table A.2: Tests of between-subjects effects

A significance value of less than 0.05 indicates a stastistically significant interaction. Therefore, only those interactions are considered for further analysis. Interactions which includes factors in a larger set of significant interactions are also not considered.