Diss. ETH No. 12906

Scheduling for Heterogeneous Opportunistic Workstation Clusters

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
MATTHIAS NEERACHER
Dipl. Inf.-Ing., ETH Zürich
born 1 February 1967
citizen of Zürich

accepted on the recommendation of
Prof. Dr. W. Fichtner, supervisor
Prof. Dr. L. Thiele, co-examiner

1998

[illegible]

Contents

List of Figures

List of Tables

Leer - Vide - Empty

Abstract

In recent years, clusters of networked workstations have become an increasingly popular source of computing resources for large, computation and memory intensive calculations. Instead of setting aside dedicated workstations for such clusters, the spare computing resources of interactively used workstations can be harnessed into opportunistic clusters. Since equipment in workstation environments is often underutilized, opportunistic clusters offer the potential of obtaining considerable computation resources at essentially no additional hardware cost.

In this dissertation, I present the design and implementation of DMW, a system to schedule a collection of interdependent computing jobs on an opportunistic cluster consisting of workstations based on a variety of hardware and software platforms, connected by a local area network and sharing a common file namespace.

Designers of distributed systems often have to choose between fully distributed communication architectures and architectures with some of the communication and decision flow concentrated in a central component. Having implemented both a fully distributed and a partially centralized version of DMW, I contrast the two approaches, concluding that the partially centralized approach offers better performance and global fairness and reduces communication overhead compared to the fully distributed approach.

Any evaluation of a scheduler for an opportunistic workstation cluster cannot rest solely on the computation performance offered to users of the scheduler: For such a scheduler to be accepted in a workstation environment, it is of primary importance that resource demands it places on a workstation never interfere with the needs of interactive users using that workstation.

To analyze this problem, I present a series of experiments documenting the resource consumption patterns of jobs scheduled by the local scheduler on a workstation. I demonstrate that CPU use of background jobs is well controlled by local schedulers and rarely interferes with interactive users, but that memory use of background jobs is a serious problem which cannot be adequately addressed by local schedulers.

Since both the resource demands of background jobs and the resource availability on the target workstations is not predictable, a scheduler must be capable of migrating running jobs to other workstations if conditions change, i.e., the job becomes too large for its current workstation, an interactive user starts using the workstation, or another, more attractive workstation becomes available. While I did not implement a migration mechanism for DMW, I present design criteria for such a mechanism. In particular, I argue that an application independent mechanism is unsatisfactory in heterogeneous environments and requires impractical amounts of disk space and communication bandwidth for the large jobs typical for the intended use of DMW, and that an application specific migration mechanism can overcome these problems.

Leer - Vide - Empty

Zusammenfassung

In den letzten Jahren sind Verbunde vernetzter Workstation-Rechner zu einem immer beliebteren Mittel zur Bereitstellung von Rechenressourcen für grosse, rechen- und speicherintensive Berechnungen geworden. Dabei müssen die Rechner für einen solchen Verbund nicht eigens bereitgestellt werden, da es möglich ist, die brachliegenden Rechenkapazitäten interaktiv genutzter Rechner in opportunistischen Rechnerverbunden zu verwenden. Weil Computer in Workstation-Umgebungen oft nur wenig ausgelastet sind, versprechen opportunistische Verbunde, beträchtliche Rechenressourcen praktisch ohne zusätzliche Hardwarekosten zu erhalten.

In dieser Dissertation behandle ich den Entwurf und die Entwicklung von DMW, einem System, das den Ablauf einer Gruppe verknüpfter Rechenaufträge auf einem opportunistischen Verbund plant. Dieser besteht aus Rechnern, die auf verschiedenen Hardware- und Softwareplattformen basieren, mit einem Netzwerk verknüpft sind, und über ein gemeinsames Dateisystem verfügen.

Die Entwickler verteilter Systeme müssen oft eine Wahl treffen zwischen einer vollständig verteilten Systemarchitektur und einer Architektur, bei der Teile des Kommunikations- und Entscheidungsflusses in einer zentralen Komponente konzentriert sind. Anhand einer vollständig verteilten und einer teilweise zentralisierten Ausführung von DMW vergleiche ich die beiden Ansätze und komme zum Schluss, dass der teilweise zentralisierte Ansatz bessere Leistung und systemweite Fairness bringt und zudem im Vergleich zum vollständig verteilten Ansatz einen geringeren Kommunikationsaufwand benötigt.

Die Beurteilung eines Schedulers (Ablaufplaners) für einen opportunistischen Rechnerverbund kann sich nicht allein auf die Leistung stützen, die den Benutzern des Schedulers geboten wird: Für die Akzeptanz eines solchen Systems in einer Workstation-Umgebung ist es von entscheidender Bedeutung, dass die Ressourcenbedürfnisse, die es an einen Rechner stellt, niemals zu Lasten eines interaktiven Benutzers des Rechners gehen.

Um dieses Problem zu analysieren, zeige ich in einer Reihe von Experimenten auf, wie sich die Ressourcenbedürfnisse von Rechenaufträgen entwickeln, die auf einem einzelnen Rechner vom dortigen lokalen Scheduler verwaltet werden. Dabei wird ersichtlich, dass der Rechenbedarf von Hintergrundberechnungen vom lokalen Scheduler gut kontrollierbar ist und kaum je Probleme für interaktive Benutzer darstellt, dass aber der Speicherbedarf dieser Berechnungen ein schwerwiegendes Problem darstellt, das vom lokalen Scheduler nicht hinreichend gelöst werden kann.

Da sowohl die Ressourcenbedürfnisse von Hintergrundberechnungen als auch die zur Verfügung stehenden Ressourcen auf einem Rechner nicht vorhersehbar sind, muss ein Scheduler imstande sein, laufende Berechnungen auf andere Rechner zu verschieben, wenn sich die Bedingungen verändern, d.h. wenn die Berechnung zu gross für ihren gegenwärtig zugewiesenen Rechner wird, wenn ein interaktiver Benutzer den

Rechner zu belegen beginnt, oder wenn ein anderer, schnellerer Rechner frei wird. Ich habe keinen Migrationsmechanismus für DMW entwickelt, diskutiere jedoch Entwurfskriterien für einen solchen Mechanismus. Insbesondere argumentiere ich, dass ein programmunabhängiger Mechanismus für heterogene Umgebungen unbefriedigend ist und dass er für die grossen Rechenaufträge, die für den Einsatz von DMW typisch sind, unrealistische Mengen an Plattenspeicher und Kommunikationskapazität benötigt. Ein programmspezifischer Mechanismus ist dagegen in der Lage, diese Probleme zu überwinden.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Prof. Wolfgang Fichtner, for his unceasing confidence in me, and for establishing such an excellent working environment. I am grateful to Prof. Lothar Thiele for co-examining this thesis and for clarifying its focus through his questions.

The work described in this thesis was carried out in an industrial cooperation between the Integrated Systems Laboratory and ISE Integrated Systems Engineering AG, supported by a grant from project 3192.2 of the Commission for Technology and Innovation (KTI).

Roland Rühl was influential in my work in many roles. As a graduate student, he arranged my employment at IIS. As the director of software development of ISE, he led the design of `GENESISe` and specification of `DMW` and convinced me to join these efforts. As the CEO of ISE, he continued lending his advice and experience to the development of `DMW`, and despite his enormous workload, he found the time to read my manuscript and improve it through numerous suggestions.

I am grateful to my colleagues at ISE, Christian Clémençon—who implemented the `DMW` front end—and Andreas Müller, for their cooperation with me during all phases of the development of `DMW` and the writing of my thesis. Their experience and commitment to quality were indispensable in getting `DMW` to work on customer sites all over the world.

I would like to thank Michael Buschauer, Felix Rauch, and Marcel Steinmann for their work on the migration mechanism for `DMW` and their inspiring suggestions for the project.

`DMW` profited a lot from internal and external user feedback. In particular, I would like to thank Thomas Feudel, Alexander Höfler, and Norbert Strecker for their constructive feedback, their unceasing insistence on obtaining a reliable system, and for never losing their sense of irony throughout the development and testing process.

Martin Gander read my manuscript at a critical development stage, and his encouraging remarks helped me overcome my writing inertia.

The Internet provided me with a 24-hour support hotline, and Chris Nandor, Tom Christiansen, and other denizens of the IRC `#perl` channel were always willing to debate some of the finer points of the English language.

Christoph Wicki and Adam Feigin kept the computer systems running, answered many configuration questions, and protected me from the wrath of other users at IIS when one of my experiments or a faulty version of `DMW` brought some workstations to their knees.

In my strained relationships with bureaucracies, paperwork, and hardware, I was grateful to have Doelf Aemmer, Christine Haller, and Hanspeter Mathys, respectively, serve as my diplomatic liaisons.

I am indebted to Terrance Crow, Robert Rogenmoser, Rumi Zahir and Reto Zimmermann for their meta-scientific advice throughout my work at IIS.

Finally, my special thanks go to my parents, Verena Welti and Hans Neeracher, and to my wife, Helene, for their love, support, and patience during the last eight years.

Chapter 1

Introduction

> Let us begin by committing ourselves to the truth, to see it like it is and to tell it like it is, to find the truth, to speak the truth and to live with the truth. That's what we'll do.
>
> Richard Nixon, *accepting the Republican nomination for president, 1968*

In the development and optimization of VLSI semiconductor fabrication processes, manufacturers are increasingly relying on *virtual fabs* for their experiments, replacing physical facilities with numerical simulations of fabrication *processes* and of electrical current flow through semiconductor *devices*.

A typical simulation consists of

- A 1D process simulation to simulate process steps before the application of the first process mask.
- A 2D or 3D process simulation to simulate the remaining process once masks are involved.
- Generating a grid for the device simulation.
- The device simulation.
- Extracting parameter values from the simulation results.

While many of these steps only take a few seconds, 2D process simulations and device simulations may take several hours and 3D process simulations may take several days. To optimize a process, simulations have to be repeated with varying parameter values.

On the other hand, the substantial resource requirements and the limited interdependence of the individual simulation steps make such simulation problems well suited for exploiting coarse grain parallelism by distributing the execution of the simulation steps among a *cluster* of networked workstations.

1.1 Workstation Clusters

With their good cost/performance ratio, steadily increasing networking bandwidth, and large selection of applications and programming tools, workstation clusters have become an increasingly popular source of computing resources in recent years. Workstations can either be *dedicated* to the task of executing computationally intensive

background applications, or an opportunistic approach can be used. *Opportunistic clusters* [PL95] attempt to make the spare computing resources of interactively used desktop workstations available for computing intensive applications.

Today's academic and industrial settings often offer the potential for large opportunistic clusters consisting of a heterogeneous collection of dozens, if not hundreds, of workstations. By their very nature, opportunistic clusters are highly dynamic since they are based on a coexistence between the batch environment and the interactive workstation users, who can regain control of "their" workstation by a single keystroke. However, equipment in workstation environments is often underutilized, with processor utilization as low as 30% [GSS89].

1.2 The DMW Scheduler

To speed up the execution of complex simulations by improving the utilization of customers' computing resources, Integrated Systems Engineering AG (ISE), a vendor of process and device simulation software, decided to integrate a distributed scheduling facility into its TCAD (Technology CAD) environment. This project was realized in an industrial collaboration between ISE and the ETH Zürich Integrated Systems Laboratory (IIS), resulting in the DMW[1] distributed scheduler implemented by me.

In this dissertation, I will discuss the design and implementation of DMW, comparing it to other schedulers described in the literature. I will cover both some aspects widely discussed in existing literature, such as scheduling policy and job transfer mechanisms, and other aspects, such as security and coexistence with software licensing enforcement mechanisms, which are not usually discussed in research systems, but are essential for operating in a commercial environment.

The overall goals of a distributed scheduler for opportunistic clusters are to execute schedules correctly and as efficiently as possible, without disturbing interactive users of the workstations on which background jobs are scheduled. I shall present empirical data to evaluate DMW against these goals.

1.3 Structure of this Dissertation

This dissertation is structured in three parts, corresponding to the planning, implementation, and evaluation components of the DMW project.

1.3.1 Objectives And Related Work

In the first part, I discuss the planning and background material for DMW: Chapter 2 highlights the objectives we had for our scheduler and sketches the computing environment in which it operates. Chapter 3 gives an overview of approaches to distributed scheduling and gives a survey of the existing literature in the field.

1.3.2 The DMW Scheduler: Design and Implementation

The next three chapters describe the implementation of the DMW scheduler: Chapter 4 discusses the implemented scheduling policies, chapter 5 discusses the underlying

[1] The acronym DMW stands for "The Devil Makes Work for Idle Hands", a traditional warning of the moral perils of under-utilized resources

mechanisms, and chapter 6 highlights some further implementation issues of interest.

1.3.3 Experiences with DMW

The dissertation concludes with an evaluation section: Chapter 7 discusses various performance measurements, proceeding from local scheduling of single processes to the execution of complex distributed simulations. Chapter 8 presents an overall assessment of our scheduling approach and gives criteria for the successful application of distributed scheduling in commercial systems.

1.3.4 Appendixes

Appendix A briefly discusses ISE's GENESISe simulation environment into which the DMW scheduler is embedded.

The mathematically oriented papers about scheduling frequently assume some familiarity with queueing theory or at least its terminology. Appendix B sketches the mathematical foundations of queueing theory.

Appendix C discusses the notation used in the diagrams of class relationships in Chapter 6.

The cited literature uses a wide variety of terminology; in the interest of consistency, I have paraphrased it using the terminology employed elsewhere throughout this dissertation and explained in the *glossary*. Some of the more commonly found original terms are preserved as cross references in the *index*.

Leer - Vide - Empty

Part I

Objectives and Related Work

Leer - Vide - Empty

Chapter 2

Scheduler Objectives

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.
"I don't much care where—" said Alice.
"Then it doesn't matter which way you go," said the Cat.
Lewis Carroll, *Alice's Adventures in Wonderland*

In this chapter, I discuss the design objectives for the DMW scheduler, starting with a specification of the task it is expected to perform and the type of jobs it executes.

While many of the systems described in the literature were designed for special purpose environments and had considerable freedom to choose (and often even modify) target hosts and operating systems (Like, for instance, the Stealth [KC91] system discussed in section 3.2.2), DMW has to serve the computing needs of an existing commercial software system on the existing general purpose hardware at customer sites. Thus, the intended target environment is a very influential part of the system specification.

The commercial nature of the jobs executed by DMW also requires some care to keep the execution of jobs within the licensing conditions in force at that site. The final section of this chapter briefly discusses those licensing related requirements.

2.1 Purpose of the Scheduler

The scheduler needs to accept a *schedule* consisting of a number of *jobs* communicating through ordinary disk files. Most of the jobs are subject to *data dependences*: Some of their input data is computed by other jobs, so they cannot start before the other jobs have finished. Furthermore, users may prefer some of the simulations to finish earlier to provide an early indication of the trends in a set of experiments. These preferences are expressed through *priority dependences*: The dependent job is not permitted to start before the prerequisite has been started. Data and priority dependences connect the jobs in a set of directed, acyclic dependence graphs.

Using available *hosts* from a set of networked computers (typically workstations with one to four processors), the scheduler needs to execute the schedule as quickly as possible, while satisfying all data dependence relations. Most of the target hosts will not be dedicated to the *background jobs* placed by the scheduler, but will also have interactive users running *foreground jobs*. For the scheduler to be tolerated by interactive users, it must avoid interfering with the performance of foreground jobs.

2.2 Work Load

The scheduler is used to run both batch oriented and interactive jobs. Many of the jobs are expected to require substantial processing time (several minutes to several hours) and memory (up to several hundred megabytes). The jobs perform file I/O and some of them use the X window system, but none of them use any other interprocess communication facilities.

2.3 Target Machines and Operating Systems

The scheduler must be able to run on at least

- Sun SPARC workstations running SunOS 4 and Solaris 2.
- Hewlett-Packard PA-RISC workstations running HP-UX 9 and later.
- DEC Alpha workstations running DEC OSF/1.
- IBM RS/6000 workstations running AIX 3.2 and later.
- Intel 80x86 workstations running Linux.
- SGI MIPS workstations running IRIX.

and must be portable to other systems with moderate effort, provided those systems meet the following requirements:

POSIX Compliance The operating system must be reasonably conformant to the IEEE 1003.1 (POSIX.1) [IEE90, Lew91] standard.

BSD Sockets The operating system must offer a TCP/IP implementation with a BSD socket [Ste90] compatible programming interface.

Common File Namespace All target systems must support a *common file namespace*, i.e., the target systems must run a distributed file system like Sun's NFS [Mic88a] or the Andrew File System [Zay91], and all files related to the jobs to be scheduled must be accessible on all hosts under the same file name.

Trusted Users The user names of scheduler users must be *trusted* on all target systems, i.e., they must be able to log into all target systems without providing a password. This is usually achieved with the `hosts.equiv` or the `.rhosts` mechanisms, both of which are not unproblematic from a security perspective.

2.4 Heterogeneity

In general, the scheduler will operate in a *heterogeneous* environment consisting of workstations equipped with a small number of microprocessors (typically 1, but occasionally also up to 4 or more), running a variety of the operating systems mentioned above and configured with widely varying amounts of main memory.

The scheduler must allow as much interoperation as possible and transparently ensure safe operation where operating systems are not interoperable, e.g., when migrating jobs. It assumes, on the other hand, that all input and output files read and written by the simulation tools are platform independent.

2.5 Scheduling Restrictions

When scheduling jobs, the scheduler has to take into account restrictions imposed by licensing agreements of the site or by the operating environment.

2.5.1 Licensing Restrictions

Licenses for the simulation tools are either *floating*, limiting the total number of copies of a program running on a group of hosts, or *node locked*, specifying a fixed set of hosts on which execution of the program is permitted.

Accordingly, the scheduler must not launch more instances of a program than the floating license permits, or launch a program on a host not covered by the program's node locked license.

2.5.2 Operating Restrictions

There may also be technical reasons for not wanting some jobs to execute on some of the hosts. For instance, a simulator or a job known to be demanding on resources might be restricted to a subset of possible hosts.

The scheduler should therefore allow the user to specify permissible hosts on a program by program or even a job by job basis.

Leer - Vide - Empty

Chapter 3

Related Work in Distributed Scheduling

> "Hold on," Hamlin says. "I'll call and make a reservation." He clicks off, leaving McDermott and myself on hold. It's silent for a long time before either one of us says anything.
> "You know," I finally say. "It will probably be impossible to get a reservation there."
> Bret Easton Ellis, *American Psycho*

In this chapter, I give an overview of the main components needed in a distributed scheduler and discuss the various approaches to them proposed in the existing literature on the topic.

In section 3.1, I shall present a taxonomy of scheduling algorithms for distributed systems. Section 3.2 discusses operational distributed schedulers described in the literature.

The remaining sections discuss related work specifically addressing the areas of scheduling algorithms (3.3), job migration mechanisms (3.4), and performance metrics (3.5).

3.1 A Taxonomy of Schedulers

Casavant and Kuhl [CK88] categorize scheduling policies into a taxonomy which is reproduced in Figure 3.1:

Global vs. Local *Local* scheduling is concerned with running jobs on a single host, typically via time-slicing. *Global* scheduling determines on which host a job runs. Since our scheduler uses the standard scheduler provided by the host's operating system for local scheduling, I will mostly discuss global scheduling.

Static vs. Dynamic *Static* scheduling determines the assignment of jobs to hosts at an early stage—often at compile or link time. *Dynamic* scheduling makes scheduling decisions based on information obtained at run time, like the load on a host. Static scheduling is not applicable to our problem domain, since it requires that the execution times of the jobs and the availability of the hosts be known a priory, neither of which is possible in our scheduler. Thus, I will not discuss static scheduling further.

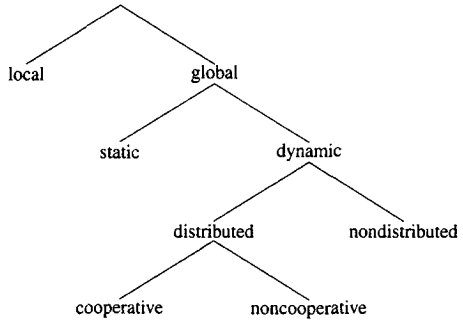


Figure 3.1: Taxonomy of scheduling algorithms (adapted from [CK88])

Distributed vs. Nondistributed In a *nondistributed* (*centralized*) dynamic global scheduler, all scheduling decisions are made on a single host, while in a *distributed* scheduler, decision making is physically distributed among the hosts.

Cooperative vs. Noncooperative Distributed dynamic global schedulers may either *cooperate* among themselves, or each host may act entirely autonomously and allocate its resources independently of the effect of its decisions on the rest of the system.

Shirazi et al. [SHK95] and Shivaratri et al. [SKS92] describe the policy employed in a dynamic global scheduler as a combination of

- An *information policy* specifying what host load information is available to the scheduler and when and how that information is transmitted to the scheduler (e.g. on demand by the scheduler, periodically, or when the host state changes significantly).
- A *transfer policy* determining under which conditions a job is transferred to a different host. This includes the question whether jobs will only be transferred before they have started (*non-preemptive* transfer), or whether they can be suspended and *migrated* to a different host while executing (*preemptive* transfer).
- A *placement policy* deciding which host in a distributed system a job will be transferred to and how to find that host (e.g. by choosing a random host, by exhaustive polling, by probing a sample of hosts).

3.2 Existing Distributed Schedulers

A considerable number of operational distributed schedulers have been described in the literature. In the following sections, I shall present a number of these systems, with an emphasis on dynamic schedulers.

Condor, described in section 3.2.1, is an academic system in wide use over the last 10 years. Condor aims to make minimal demands on the operating environment and

to avoid interfering with the work of interactive users, employing *job migration* for the latter objective.

The Stealth system, described in section 3.2.2, uses *preemptive allocation* of CPU, memory, and I/O resources to avoid interfering with interactive users.

PBS (section 3.2.3) and MESSIAHS (section 3.2.4) are two systems emphasizing flexibility, and, in the case of MESSIAHS, autonomy.

3.2.1 Condor

The Condor scheduling system [LLM88, LBRT97] operates in a workstation environment, identifying idle workstations to schedule background jobs on them. Condor employs job migration to keep the interference of background jobs with the activities of interactive users to an absolute minimum.

As soon as interactive user activity is detected on a workstation with background processes running, the background jobs are immediately suspended. If the interactive activity persists more than a few minutes, the background jobs are migrated to another host. The originating host is then kept free of background jobs until there has been no interactive activity for a significant amount of time. Since Condor is designed to accommodate a wide range of administrative policies, tolerable levels of background vs. interactive activity can be defined using various criteria, e.g., the time of day or the number of keystrokes at the console during a period.

Usage of Condor varies widely, with heavy users requiring lots of computing capacity, and light users only requiring remote computing capacity occasionally. To provide fair access to these user groups, Condor employs the *Up-Down algorithm*: A counter is maintained for each user, which is increased when the user has been allocated remote computing capacity, and decreased when the user has tried to allocate remote capacity but has been denied. Users are then prioritized, with lower counter values receiving higher priorities, and jobs of lower-priority users are preempted to make room for the jobs of higher priority users.

Condor employs a compromise between distributed and nondistributed scheduling, with a central coordinator allocating computing capacity to schedulers running on each node and the nodes scheduling their assigned capacity among their own jobs.

When a process is migrated to a new host, Condor maintains a *shadow process* on the original host. System calls on the new host are redirected into RPC calls communicating with the shadow process. While this approach maintains network transparency without requiring a common file namespace or remote login capability from the hosts, it introduces *residual dependencies* at a considerable potential cost in performance and reliability.

3.2.2 Stealth

Like Condor, the Stealth system [KC91] is built on the principle that background jobs are not to interfere with interactive use. Instead of primarily relying on process migration to achieve that goal, Stealth is using a modified Mach [ABG+86] kernel as the local scheduler to insulate foreground jobs from the demands of background jobs through *priority resource allocation*:

- No background job will receive CPU time as long as a foreground job is eligible to run.

- No background job will be able to claim a page of physical memory as long as a foreground job is short of memory.
- No background job will be able to claim a disk buffer as long as a foreground job needs more disk buffers.
- No disk I/O for a background job is done as long as there are pending I/O requests from foreground jobs.

With this approach, the authors report that Stealth allows background jobs to utilize up to 90% of available CPU capacity while their influence on the performance of foreground jobs is negligible regardless of load conditions.

Stealth employs a fully distributed global scheduler and is capable of migrating processes, although this is rarely necessary in this system.

3.2.3 PBS

PBS, the Portable Batch System [Hen95], is designed to flexibly support a variety of batch scheduling policies on a mixture of hardware configurations. The PBS implementation is distributed across four types of daemon processes:

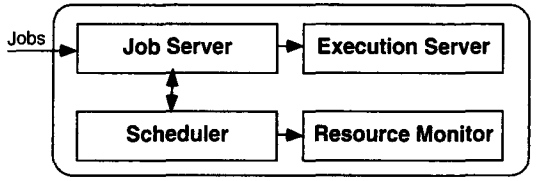
- *Job Servers* own and manage jobs and job queues, are a central collection point for jobs, and a focal point for client communication. Job servers transfer jobs to their associated Execution Servers or to other Job Servers.
- *Execution Servers* control all batch jobs on a host. They start jobs, monitor and control their resource usage, and clean up after they complete.
- *Resource Monitors* gather resource availability and usage information about the host on which they run.
- *Schedulers* obtain information about jobs ready to run from their Job Server and about resource availability from the Resource Monitors. They then direct the Job Server as to what, if any, action should be taken.

This distribution of responsibilities among servers allows different organizations to be used e.g. on supercomputers (where all four daemons might run on the same system) and on workstation clusters managed as a single subsystem (where each host would run its own Execution Server and Resource Monitor, but only one Job Server and Scheduler would control the entire subsystem).

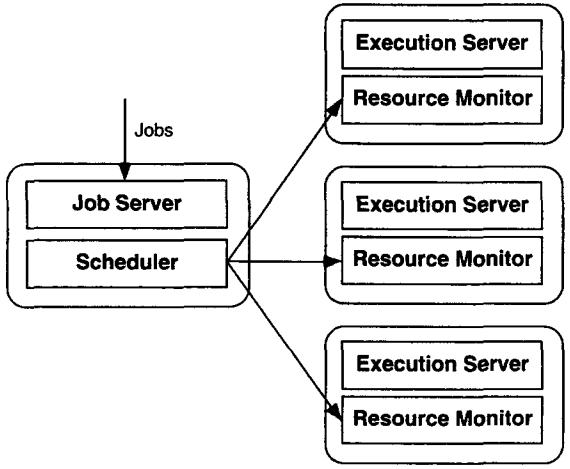
PBS emphasizes easy development of new scheduling policies, offering the possibility to

- write scheduling scripts in *BASL*, a BAtch Scheduling Language designed for PBS.
- write schedulers in Tcl (Tool Command Language)[Ous94], a standard scripting language.
- write schedulers in C or another traditional programming language, using the direct API supplied for PBS.

Henderson [Hen95] shows that this flexibility has only a negligible cost in terms of performance.



(a) Single Host



(b) Cluster

Figure 3.2: PBS daemons on different system configurations

3.2.4 MESSIAHS

The MESSIAHS system [Cha93] provides automated support for task placement in heterogeneous distributed systems with a particular emphasis on preserving *autonomy* in various forms:

design autonomy Each host's hardware and operating system may be designed (or purchased) independently of the architecture of the other hosts.

communication autonomy Each host can make independent decisions about what information to release, what messages to send and reply to, and when to send them.

administrative authority Each host sets its own resource allocation policies, independent of the policies of other systems.

execution autonomy Each host decides whether it will honor a request to execute a job and has the right to stop executing a job it had previously accepted.

Based on these principles, MESSIAHS provides a flexible framework which can be adapted to a variety of static and dynamic scheduling algorithms. While mostly preserving all of the above autonomies (obviously, preserving e.g. total communication autonomy is unrealistic for a distributed algorithm), MESSIAHS produces quite efficient schedules.

3.3 Related Work: Scheduling Algorithms

This section discusses various approaches taken in scheduling algorithms. Zhou and Ferrari (section 3.3.1), Theimer and Lantz (section 3.3.2), and Shivaratri, Krueger, and Singhal (section 3.3.3) discuss information and placement policies in traditional, non-adaptive algorithms.

In contrast, Ferguson et al. (section 3.3.4) propose an adaptive, decentralized approach based on concepts drawn from microeconomics, and Kipersztok and Patterson (section 3.3.5) present a scheduler using a placement policy based on fuzzy logic.

3.3.1 Zhou and Ferrari

Zhou and Ferrari [ZF87] studied five different sender initiated (see section 3.3.3) scheduling algorithms (some of them proposed by Eager et al. [ELZ86]) as shown in Figure 3.3:

DISTED On each host, a *Load Information Manager* (LIM) computes a load index. If the value has changed significantly, it is broadcast to all other LIMs. Each scheduler attempts to place new jobs with the local LIM. If the local load is above a threshold, the LIM places the job instead on the host with the lightest load.

GLOBAL A central LIM receives the load information from all other LIMs and broadcasts a vector containing all load information.

CENTRAL The central LIM receives the load information as in GLOBAL, but handles all job placement requests in the system.

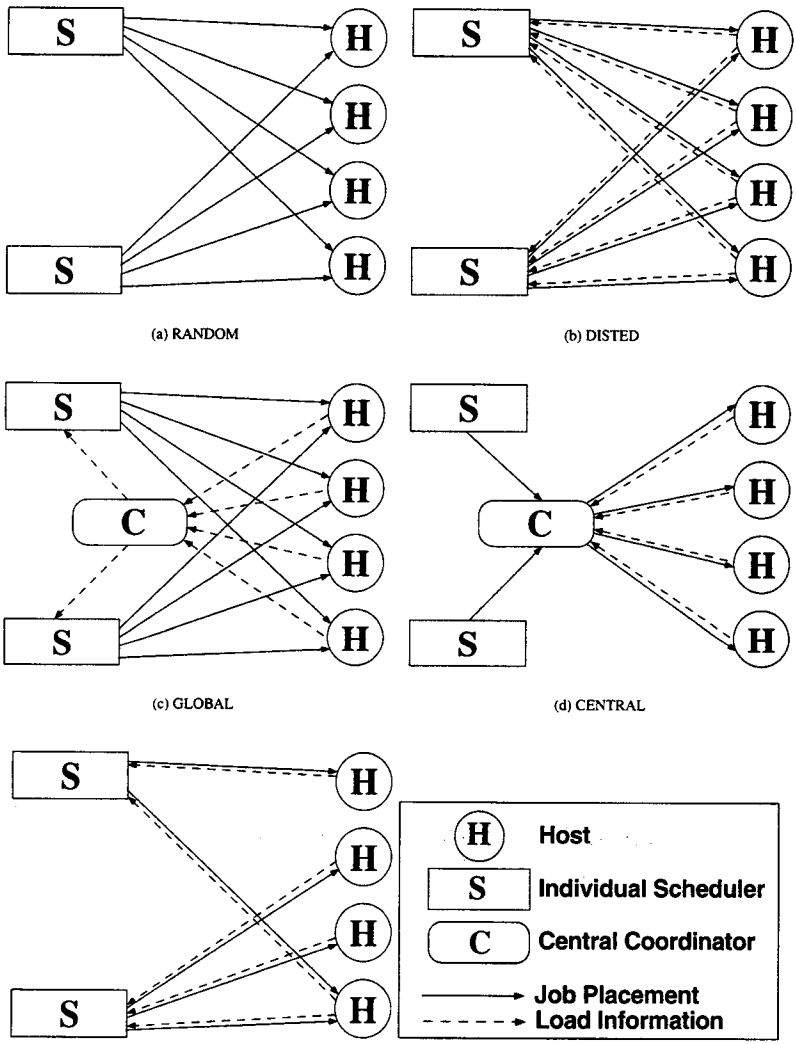


Figure 3.3: Schedulers described by Zhou and Ferrari [ZF87]

LOWEST Load information is only sent out on demand: To place a job, the local LIM polls a relatively small, randomly chosen, subset of the other LIMs and places the job with the best host of this subset.

RANDOM No load information is exchanged; jobs are transferred to a randomly chosen host.

Evaluating implementations of these algorithms on a network of six diskless Sun-2 workstations supported by a central file server, the authors found that

- All five algorithms provided significant performance improvements over running the workload on a single processor.
- RANDOM with its arbitrariness and DISTED with its N^2 communications structure performed considerably worse than the other three algorithms.
- LOWEST provided the best performance.

In simulations of systems with up to 49 hosts, GLOBAL, CENTRAL, and LOWEST scaled up well and remained comparable in performance while DISTED performance deteriorated considerably with increasing system size.

3.3.2 Theimer and Lantz

Theimer and Lantz [TL88] studied a similar set of algorithms as Zhou and Ferrari, but on a considerably bigger network consisting of about 70 hosts running the V Operating System [Ber86], basing their system on the multicasting operations and the transparent remote execution facility provided in V.

The authors modelled and implemented a centralized algorithm similar to Zhou and Ferrari's CENTRAL and a decentralized algorithm similar to DISTED and LOWEST. Both algorithms were refined with the introduction of an *update group* and an *idle group*, both apparently implemented as V multicast groups.

Machines join the idle group if they have idle resources available for remote execution requests and leave it if they are entirely busy. Machines in the idle group send infrequent updates of their load status. A *cutoff value* is determined based on the demand for remote scheduling and the known average load. This cutoff value is periodically multicast to the idle group, and hosts whose load average is below that cutoff value join the update group. Machines in the update group are queried about their load once every 10 seconds.

For their distributed algorithm, Theimer and Lantz point out that algorithms such as LOWEST rely on statistical assumptions of a homogeneous workload, and they argue that

> ... an environment consisting principally of personal workstations tends to produce a load distribution that can vary greatly in magnitude over time and is *not* homogeneous in nature. Hosts may be idle or running interactive applications or compute-intensive batch jobs or both. "Clusters" of hosts may be idle when events such as group meetings occur.

So, instead of sampling only a subset of hosts as in LOWEST, load queries are sent to all hosts in the update group (comparable to DISTED), but only very lightly loaded hosts answer immediately.

Other hosts delay their responses by an interval

$$\Delta = \delta\lambda r,$$

where δ is a standard delay interval, λ is the load of the host normalized to the interval $[0, 1]$, and r is a uniformly distributed random value. The scheduler discards all but the first n responses, where n is very small compared to the size of the update group.

This algorithm has several favorable properties:

- The load correlated delay makes less loaded hosts more likely to be among the first n to respond.
- The random factor r improves the chance that even when the load on all hosts is high, some hosts will respond quickly.
- The authors report that the performance of this selection scheme is within 2% of the "perfect" centralized scheme if $n = 3$ and within 1% if $n = 6$.

However, replies are still sent by all hosts in the update group, which takes up network bandwidth and ultimately limits scalability of this algorithm. Furthermore, redundant replies may fill up the available buffers of the network software, causing other, useful network packets to be discarded.

The distributed algorithm also has to handle the problem of *contention*, that several schedulers decide simultaneously to submit jobs to the same host. The solution proposed by the authors is to include in the execution request the load that the scheduler expects to see on the host. If the actual load is significantly higher, either the target host selects an alternate host by repeating the selection procedure, or the scheduler already sends an alternate host with the execution request and the target host simply forwards the request there.

While Theimer and Lanz conclude that their centralized system performs faster and scales to a higher number of hosts, they point out that a decentralized system is simpler, as it does not have to include recovery procedures for a failure of the central host. They therefore recommend a decentralized scheduler and

> ... that one should switch to a centralized design when scalability beyond a few hundred hosts becomes a significant issue, or when other issues (such as global fairness or network management) come into play.

3.3.3 Shivaratri, Krueger, and Singhal

Shivaratri et al. [SKS92] studied various distributed load sharing algorithms to transfer jobs from *sender hosts* with higher load averages to *receiver hosts* with lower load averages. They classified the algorithms they studied into:

Sender Initiated Heavily loaded hosts transfer newly arriving jobs to other hosts in the system. As discussed in section 3.3.1, there are various approaches to determining the receiver, from exhaustive polling to an entirely random choice.

Sender initiated algorithms have the advantage that they do not need to employ job migration. However, at high system loads, they tend to cause *instability*: The polling and transfer activity puts an increasing strain on the system while benefits diminish.

Receiver Initiated Lightly loaded hosts migrate running jobs from other hosts in the system.

These algorithms work well at high system loads, but they have to rely on job migration, which tends to be more complex to implement than job placement. Furthermore, if only a few hosts in the system are heavily loaded, receiver initiated algorithms can easily miss them.

Symmetrically Initiated Both of the above mechanisms are employed simultaneously.

The sender-initiated component is successful at finding underloaded nodes at low system loads, and the receiver-initiated component is successful at finding overloaded nodes at high system loads. However, symmetrically initiated algorithms also combine the disadvantages, requiring job migration and causing instability at high loads.

Adaptive Symmetrically Initiated To avoid instability, the adaptive symmetrical algorithm keeps track of past polling responses to classify other hosts as *senders* (high load), *receivers* (low load), and *neutral* (average load). The sender initiated component only polls hosts classified as receivers, and thus reduces activity at high loads. The receiver initiated component tries first senders, then neutral hosts, and finally receivers.

While this approach causes extra activity at low system loads, it keeps classifications up to date and the resources consumed do not adversely affect performance since extra processing capacity is available anyway.

Adaptive Sender Initiated This algorithm both classifies hosts and keeps track of how the own host is classified by other hosts. The receiver initiated component does *not* transfer jobs, but solely informs hosts which have the current host classified as a sender or neutral that the host has become a receiver.

Thus, this algorithm combines the advantages of the adaptive symmetrical and sender initiated algorithms, avoiding both instability and the need for job migration.

The authors simulated these algorithms for a system consisting of 40 hosts, assuming independently exponentially distributed task interarrival times and service demands (M/M) and comparing the algorithms with a system doing no load distributing ($40 \star M/M/1$) and a (theoretical) system doing perfect load distributing without communication overhead ($M/M/40$).

Varying the overall system load for a *homogeneous* system, where jobs were equally likely to arrive at any of the hosts, the authors found that

- All of the algorithms studied provided a substantial performance improvement over a system without load sharing.
- Sender initiated algorithms (the authors simulated the algorithms called RANDOM and LOWEST by Zhou and Ferrari) were better than a receiver initiated algorithm (with random probing) at low system loads, and the receiver initiated algorithm was better at high system loads.
- The adaptive symmetrically initiated algorithm performed best under all loads.

- The adaptive sender initiated algorithm was also better than any of the nonadaptive algorithms, except at very high system loads. While this algorithm did not quite reach the performance of the adaptive symmetrically initiated algorithm, it worked purely with job placement and did not require job migration.

In a second comparison, the authors simulated a *heterogeneous* system, where they varied the number of nodes where jobs arrived, while keeping the overall system load constant at 0.85 (i.e. the incoming workload was 85% of the total CPU capacity of the system), and found that:

- A system without load sharing would become instable when the number of load generating nodes was less than or equal to 33.
- The receiver initiated algorithm would become instable for 35 or fewer load generating nodes, as at the high system load simulated, random probes were quickly losing effectiveness at locating a sender node when only a few nodes were senders.
- The LOWEST sender initiated algorithm would become instable for 25 or fewer load generating nodes as the remaining load generating nodes were receiving jobs very quickly and could not locate receiver nodes quickly enough, with many of the random polls for load information failing to locate suitable receivers.
- The symmetrically initiated and adaptive sender initiated algorithms would become instable for 15 or fewer load generating nodes. Both algorithms were able to transfer jobs more quickly and effectively than the above algorithms, but eventually, the symmetrically initiated algorithm would fail because of overly ineffective polling, and the adaptive sender initiated algorithm would fail because its inability to perform job migration prevented it from using the receivers it found.
- Surprisingly, the simple RANDOM sender initiated algorithm, while never performing very well, was able to avoid instability even at extreme levels of heterogeneity.
- However, the adaptive symmetrically initiated algorithm performed better than all other algorithms, and in fact, it performed even better with increasing heterogeneity, as the classification of hosts as senders or receivers would change less frequently.

Thus, the authors concluded that the adaptive symmetrically initiated algorithm provided the best performance of all algorithms they discussed. However, it should be noted that all the algorithms they studied were *fully distributed* without a centralized component. The authors rejected approaches like Zhou and Ferrari's GLOBAL and CENTRAL because of concerns about

Robustness when the host running the central component would fail.

Performance when the central component would become a performance bottleneck in systems with many hosts.

These concerns are not shared universally. Litzkow et al. [LLM88] estimated even in 1988 that the central coordinator in Condor would have been able to handle up to 100 hosts without even having a noticeable effect on the perceived performance of

interactive users at that host, and improvements in hardware since then have further raised that limit (Livny reports in a 1997 paper [LBRT97] that the Condor coordinator now manages more than 300 hosts).

3.3.4 Ferguson et al.

As an alternative to the increasing complexity of traditional, cooperation based scheduling policies, Ferguson, Yemini, and Nikolau [FYN88] propose an approach based on concepts drawn from *microeconomics*: Jobs enter the scheduling system with some initial amount of money and seek to purchase CPU time on some host and network capacity between that host and the originating host of the system. Machines auction off their CPU time and link capacity to the highest bidder and advertise price changes to their neighbors. Jobs seek to be served as cheaply as possible; hosts try to earn as much money as possible; all agents in the system strive to maximize their individual good, without concern for the entire system.

In simulations with varying job bidding policies and host auction methods, Ferguson et al. demonstrate that their approach

- Is usually as good, and sometimes much better than a traditional approach in a system based on point-to-point links.
- Is inherently modular and decentralized and thus less complex than the alternatives.
- Is inherently stable: Even though there is no arbitrary limit on the number of migrations a job can perform, economic considerations cause jobs to migrate less as the increasing demand causes link prices to rise.
- Can accommodate a diversity of coexisting policies, as long as there is a common market mechanism.

The approach described by Ferguson et al. assumes a network with point-to-point links and jobs whose CPU demand is known in advance. In our system, these assumptions do not hold, and it is not clear how well their approach would work in this case.

3.3.5 Kipersztok and Patterson

Kipersztok and Patterson [KP95] designed a control system that uses *fuzzy logic* to prioritize the allocation of parallel jobs to, and their suspension from, a cluster of networked workstations. Their system acts as the policy component of some existing network queueing system (the authors used DQS [Gre94] for their research).

Newly arriving and suspended jobs get assigned priorities and queued according to their priorities by the *Fuzzy Controller* (FC). Jobs are then scheduled until the FC determines that no further jobs should be started at the moment. Jobs running on the cluster are then also assigned priorities, low priority jobs are suspended to make room for high priority jobs still in the queue until no job in the queue has a higher priority than a job running on the cluster.

Priorities for jobs are determined for a number N_c of cluster parameters, where each cluster parameter has a *weight* w_i and the job has a *susceptibility* s_{ij} for each parameter. The priority is then defined as

$$p_j = (1/N_c) \sum_{i=0}^{N_c} 1 - w_i s_{ij}$$

such that, the higher the weight of a cluster parameter and the higher the susceptibility of the job to that parameter, the lower the priority of the job. The highest priorities are assigned to jobs that are least sensitive to the parameters which characterize the cluster, and during times when the cluster is least loaded and thus its parameters are least important.

The parameters w_i and s_{ij} are each computed using a *fuzzy control* algorithm. The approach of separately treating the contribution of each cluster parameter to the priority has the advantage of keeping the number of rules for each of the fuzzy control systems small and allowing the system to be extensible, adding 2 extra control systems for each additional parameter.

Each fuzzy control system is a function obtained by

- Identifying the fuzzy input and output variables.
- Partitioning the domain of each variable into fuzzy, partially overlapping sets and define membership functions to each of the sets. As an example, a network that is 20% loaded could be assigned a 50% probability of being in the "low load" set and a 10% probability of being in the "medium load" set.
- Mapping inputs to outputs using appropriate heuristics.

Kipersztok and Patterson implemented a version of their system with two parameters: Number of available hosts and network load. The four fuzzy control algorithms guided by the following heuristics:

- The *Weight Factor for Number of Machines* decreases when many hosts are available; it increases when either few hosts are available or when the network is highly loaded (as a parallel algorithm requiring many hosts will require more network resources).
- The *Weight Factor for Network Load* increases and decreases with the network load itself.
- The *Susceptibility Factor for Number of Machines* increases as the number of hosts a job needs increases.
- The *Susceptibility Factor for Network Load* increases both with the communication requirements of a job and with the number of hosts the job requests.

The authors present a small example comparing standard DQS scheduling (which is simply FIFO) with scheduling of DQS augmented with their fuzzy controller and show that the run time under pure DQS is about 30% longer in this example. While the system as implemented is designed for parallel jobs, the principles for fuzzy control described would apply to sequential jobs as well.

3.4 Related Work: Job Migration Mechanisms

To support preemptive job transfers, schedulers need a migration mechanism. Eskicioğlu (section 3.4.1) discusses a number of different mechanisms for migrating jobs between hosts running the same operating system on the same hardware. Roush and Campbell (section 3.4.2) present a mechanism based on the techniques discussed by Eskicioğlu, but running much faster.

Theimer and Hayes (section 3.4.3) present an approach to migrate jobs between different operating systems and hardware platforms by capturing the state of programs to be migrated in a high level language program and recompiling that program on the new host.

3.4.1 Eskicioğlu

Eskicioğlu [Esk89] discusses the design of job migration facilities. He identifies the time spent in transferring the virtual memory of a job as the dominant cost of migration and discusses four different transfer methods:

Entire Virtual Memory Transfer This is the most primitive scheme: The job on the source host is suspended, the *entire* virtual memory is transferred to the destination host along with the rest of the job state and finally, the job is allowed to resume execution on the destination host.

While this scheme is simple and can be implemented with minimal support from the operating system, the execution of jobs usually stops for many seconds during a migration, which makes the scheme unattractive for real-time or interactive applications. Zayas [Zay87] reported that on average almost 60% of the virtual memory of migrated jobs is never referenced after a migration, so transferring the entire memory is wasteful.

Pre-copying After the migration decision has been made, the source host starts transferring the job state to the destination host while the job still runs. Memory pages modified during that time are then transferred again and this process is repeated until either only a few modified pages remain or the number of modified pages stabilizes. Finally, the job is suspended and the remaining modified pages are transferred.

This scheme substantially reduces the time a job is stopped during migration, but it requires operating system support to identify modified pages. Furthermore, pre-copying requires even more communication resources than entire memory transfer.

Copy on Reference This scheme transfers all of the state information except the virtual memory. Virtual memory pages are then only transferred when they are referenced by the job on the destination host.

This scheme minimizes the number of pages transferred, but creates a residual dependency on the source host and is slower because all page faults require communication with the source host. Implementing copy on reference generally requires modifications in the memory management subsystem of the underlying operating system.

Enhanced Copy on Reference Instead of keeping the virtual memory image, the source host writes modified pages to a file server and the destination host requests pages from the file server.

This scheme avoids the residual dependency, but page faults may be slower than with algorithms communicating directly with the source host. Implementing enhanced copy on reference may require substantial changes to the underlying operating system's memory management and disk organization, e.g. to allow one host to write to the swap disk space of another host.

3.4.2 Roush and Campbell

Roush and Campbell [RC96] discuss the *Freeze Free* algorithm, a process migration algorithm they designed to minimize process migration latency. *Freeze Free*, a refinement of the Copy on Reference and Enhanced Copy on Reference techniques described in section 3.4.1, migrates a process by performing the following steps on the old host:

- The migrating process is suspended.
- The process control and execution state are sent to the new host.
- The current *code page* (determined from the program counter), *heap page* (determined from a heuristic search in the instruction stream for a load or store instruction), and *stack page* (determined from the stack pointer) are sent to the new host.
- The rest of the stack is sent to the new host.
- The communication and file state are sent to the new host.
- The dirty memory and file cache pages are flushed to the file server.
- A message is sent to the new host indicating that the transfer is complete.
- If an acknowledgment of migration acceptance has been received in the meantime, the process status on the old host is discarded.

The new host sends a message indicating that it accepts the migration upon receiving the process control and execution state, but the old host does not wait on this message. As soon as the current stack page is received, the process is resumed on the new host, which can run since it has its essential pages in memory. The new host knows which virtual memory pages are dirty at the time the migration started. Requests for these pages are sent to the old host until all of them have been flushed to the file server, while requests for clean pages are sent to the file server. After the old host has sent the message indicating that all dirty pages have been flushed, all paging requests are handled by the file server.

Freeze Free also minimizes the period during which *message processing* for the migrating process is blocked by employing the mechanisms of the underlying CHOICES [CIRM93] operating system and *x*-Kernel [HP91] network communications architecture:

- After migration starts, incoming messages are retained but not yet delivered to the process.

- After the transfer of the communication status begins, incoming messages are rejected with a message asking the originator to update its location information for this process and retransmit the message to the new host.
- As soon as the transfer of the communication status is complete, the new instance of the process enables message handling.

Roush and Campbell provide measurements to demonstrate that *Freeze Free* drastically reduces migration latency: Between SPARCstation 2 computers connected with a 10 Mb/s Ethernet, migrating a process has a latency of only 14ms, compared to latency times of 250ms to 750ms for all other process migration systems described in the literature. In a detailed breakdown of migration costs, the authors show that the time taken for the *Latency Period* is dominated by the cost of gathering the process state information and traversing the process structures. The cost of the *Demand Paging Period*, on the other hand, is dominated by communication costs and thus will profit from faster networking hardware.

Like the enhanced copy on reference scheme described by Eskicioğlu, the implementation of freeze free assumes substantial control over the underlying operating system's memory and swap space management, which makes it difficult to implement in existing general purpose systems such as the hosts on which the DMW scheduler is designed to run.

3.4.3 Theimer and Hayes

Most process migration systems are limited to migrating jobs between machines with nearly identical hardware and operating systems. Theimer and Hayes [TH91] propose to overcome that restriction by capturing the state of the migrating process in a machine independent high-level or intermediate-level language program and recompiling and running that program on the target machine.

As an example, consider a call stack where procedure A calls procedure B, which in turn calls procedure C. The migration mechanism will create procedures B1 and C1, which reflect the state of B and C at migration time. If B is, e.g.:

```
procedure B:
   s0;
   for (i=0; i<n; i++) {
      s1;
      C();
      s2;
   }
   s3;
```

and the loop has executed fully four times, B would be transformed into a procedure B1:

```
procedure B1:
   Init_Local_Variables; /* Includes a statement i := 4; */
   C1();
   goto L;

   /* Copy of B, with label */
```

```
    s0;
    for (i=0; i<n; i++) {
       s1;
       C();
L:
       s2;
    }
    s3;
```

This procedure initializes its local variables, and calls other modified procedures further up the stack. When C1 returns, it will have done all the work the fifth invocation of C was committed to do before the migration, and the `goto` will cause resumption of the work B was committed to. Note that each modified procedure is called only once: Within the loop, C, not C1, is still called.

The ability to capture the state of a process this way imposes some requirements on the language and implementation used, notably:

- Each program must contain a number of *migration points*, such that whenever migration is requested, the program will reach a migration point within a limited amount of time.
- At each migration point, it must be possible to translate the physical machine state into an abstract, machine-independent program. For that purpose, the source level symbol tables generated by compilers for symbolic debuggers can be used.
- Each invocation of a procedure containing migration points must itself be a migration point, as the state of all procedures on the call stack has to be capturable.
- The heap must be in a consistent state, all pointers can be interpreted correctly, and every field of every heap object must be defined unambiguously.

Most programming languages in common use are not sufficiently well-defined to meet the above criteria, which has hindered the practical application of Theimer and Hayes' proposal. In recent years, however, the Java programming language, which has a well-defined execution model, heap data model, and abstract machine, has gained widespread acceptance, and heterogeneous migration work based on Java is currently in progress [PP97].

3.5 Related Work: Performance Metrics

Part of an information policy is the choice of a suitable *load index* to describe the state of a system. Ferrari and Zhou in section 3.5.1 study various load indices and the impact they have on scheduling performance.

Devarakonda and Iyer (section 3.5.2) developed a model to predict the resource requirements of jobs to be run based on past executions of the same program.

Wang and Morris (section 3.5.3) propose a quality metric to characterize the performance and fairness of a scheduling algorithm and apply it to various classes of scheduling algorithms.

3.5.1 Ferrari and Zhou

Ferrari and Zhou [FZ88] studied the impact that the choice of a load index had on scheduling performance in otherwise identical schedulers. They identify a number of desirable properties for a load index. A good load index should:

- reflect the user's qualitative estimates of the current load on a host;
- be usable to predict the load in the near *future*, since the response time of a job will be affected by the future load rather than the present one;
- be relatively stable over time without excessive fluctuations;
- have a simple relationship with the *performance index*, so that its value can be easily translated into the expected performance of a job transferred to the host.

The authors chose a scheduling algorithm with sender initiated job placement and centralized load information gathering (discussed in Section 3.3.1 under the name GLOBAL) and combined it with a number of load indices, varying four factors:

- The *load indices* themselves. The authors used
  - The instantaneous *CPU queue length.*
  - The exponentially smoothed CPU queue length.
  - The sum of the averaged CPU queue, I/O queue (for both file and paging/swapping I/O), and memory queue[1].
  - The average *CPU utilization* over a recent period.

  Resource queue lengths q_i were sampled at 10 ms intervals, averaged over one-second intervals and exponentially smoothed over the last T seconds:

$$\begin{aligned} Q_0 &= 0, \\ Q_{i+1} &= Q_i(1-e^{-T}) + q_{i+1}e^{-T} \end{aligned}$$

- The interval T over which averages were computed.
- The interval P at which load information updates were provided to schedulers.
- The *workload* used for measurements. Three different synthetic workloads were constructed out of common UNIX commands and sleep commands:
  - The *light* (L) scripts generated a CPU utilization of 30%–45%.
  - The *moderate* (M) scripts generated a CPU utilization of 60%–70%.
  - The *heavy* (H) scripts generated a CPU utilization of 70%–85%.

[1] The authors identified a number of places in the kernel where processes were queueing up for memory resources such as buffer space and memory pages and combined all of these queues into a "memory queue"

Tests were conducted on 6 workstations; the canonical workload was to have two hosts running each of the three types of loads (2L, 2M, 2H); the alternate workload studied consisted of all hosts running moderate loads (6M). The average process in these workloads took 7.45*s* to execute.

A first test series used eleven different load indices with the canonical work load:

- The instantaneous CPU queue length.
- The CPU queue length averaged over 1s, 4s, 20s, and 60s.
- CPU, I/O, and Memory queue lengths averaged over 4s, 20s, and 60s.
- The standard 1 minute UNIX *load average*.
- CPU utilization averaged over 10s and 60s.

The results from this series supported a number of conclusions:

- All of the indices improved response time by 20%–40%.
- CPU queue length based indices performed significantly better than CPU utilization based indices. The reason for this was probably that when hosts were heavily loaded, CPU utilization approached 100% and ceased to reflect the exact load. CPU queue lengths, however, did not decrease in accuracy at heavy loads.
- Indices based on a combination of several resource queue lengths did not outperform indices based on CPU utilization alone.
- Averaged indices performed better than instantaneous indices, but performance decreased significantly when the averaging interval T was 20s or longer.

A second series of experiments was conducted using the more balanced 6M workload. While the relative ranking of load indices remained quite similar, performance gains were only 2%–26% and poor load indices yielded little or no performance improvements.

In a final series of experiments, the authors varied the update interval P. As expected, frequent updates ($P < 10$s) worked best, but had a high communication cost. Even when updates were only sent every minute, the system still worked significantly better than a system without load balancing.

In applying the results of Ferrari and Zhou to the present work, it should be noted that the workloads they studied were very different from the ones that our scheduler has to handle: The average running time of jobs in our system is several orders of magnitude longer than the 8 seconds in their workloads, so it is plausible that longer averaging and update intervals work better in our system than in theirs.

3.5.2 Devarakonda and Iyer

Devarakonda and Iyer [DI89] argued that with proper initial placement of jobs, considerable savings in the cost of job migrations and global system status updates could be achieved. In order to aid initial placement decisions, they developed a statistical approach for predicting the CPU time, file I/O, and memory requirements of a job at creation time, based on the identity of the program to be executed.

Analyzing the resource usage of more than 65000 jobs gathered during one week of operation on a VAX11/780 running BSD UNIX, the authors partitioned the jobs into seven statistical *clusters*. For each program (such as `cc`, `ls`), they then built a *transition model* of the probabilities p_{ij} of the next execution of a program to require resources characteristic for cluster j given that its last execution was characteristic for cluster i.

Predicting the resource requirements on jobs based on such transition models, Devarakonda and Iyer achieved a good correlation between predicted and actual requirements: For all three resources, correlation coefficients were more than 0.8 and over 80% of prediction errors were less than 0.5 standard deviations in magnitude. However, given that their work was performed on a general purpose workload consisting of rather small and short-running jobs, it is doubtful whether their model would be directly applicable to our problem domain.

3.5.3 Wang and Morris

Wang and Morris [WM85] propose a taxonomy of load sharing algorithms based on

- The distinction between *source-initiative* strategies, where schedulers decide on the hosts to place their jobs on and *server-initiative* strategies, where hosts decide on the schedulers from which they accept jobs.
- The amount of information used by the strategy.

Their taxonomy is summarized in table 3.1. The authors note that other literature refers to level 1 algorithms as *static*, to level 5 algorithms as *dynamic*, and sometimes to level 3 algorithms as *semidynamic*.

To compare the performance of these algorithms, Wang and Morris introduce a performance metric they call the *Q-factor* (quality of load sharing factor). They define the Q-factor of an algorithm A as:

$$Q_A(\rho) = \frac{\text{mean response time over all jobs under FCFS}}{\max_i \{\text{mean response time for } i\text{th scheduler under algorithm } A\}}$$

where ρ, the aggregated utilization of the system, is defined as

$$\rho = \frac{1}{K\mu} \sum_{i=1}^{N} \lambda_i,$$

with N being the number of schedulers, K the number of hosts, μ^{-1} the mean service time of a job, and λ_i the job arrival rate at the ith scheduler.

The Q-factor, which is usually between zero and unity (but can be larger, e.g. for SJF scheduling) both characterizes the general performance of a system and exposes any bias in performance against arrival at a particular scheduler.

Wang and Morris then proceed to analyze 10 of the above algorithms for a system with N schedulers and K hosts under various assumptions (in all but the last case, communication overhead is assumed to be negligible):

1. Poisson arrivals, exponential service time distribution (M/M).
2. Poisson arrivals, deterministic (constant) service time distribution (M/D).
3. Case 2) with $K \to \infty$.

Level	Source-Initiative	Server-Initiative
1	host = f(sched) e.g. source partition	sched = f(host) e.g. server partition
2	host = f(sched, ω) e.g. random splitting	sched = f(host, ω) e.g. random service
3	host = f(sched, ω, sequence) e.g. cyclic splitting	sched = f(host, ω, sequence) e.g. cyclic service
4	host = f(sched, ω, sequence, host idle status) e.g. cyclic splitting preferring idle hosts (not analyzed)	sched = f(host, ω, sequence, scheduler idle status) e.g. cyclic service skipping idle schedulers (not analyzed)
5	host = f(sched, ω, sequence, host load) e.g. join shortest queue (JSQ)	sched = f(host, ω, sequence, scheduler queue length) e.g. serve longest queue (SLQ)
6	host = f(sched, ω, sequence, host load, latest job departure from host) e.g. JSQ using latest departure time to break ties (not analyzed)	sched = f(host, ω, sequence, scheduler queue length, latest job arrival at scheduler) e.g. first come first served (FCFS)
7	host = f(sched, ω, sequence, host load, departures of completed and remaining jobs from host) e.g. FCFS (not analyzed)	sched = f(host, ω, sequence, scheduler queue length, job arrivals and execution times at scheduler) e.g. shortest job first (SJF)

Table 3.1: Taxonomy of load sharing algorithms, adapted from [WM85]. ω stands for a randomly generated parameter.

4. Poisson arrivals, hyperexponential service time distribution (M/H).

5. Batched Poisson arrivals, exponential service time distribution ($M^{[x]}/M$).

6. Case 1) with non-negligible communication overhead.

Where possible, the authors used an exact analysis of the algorithms, while in other cases, they had to resort to simulations. They concluded that

- At least in the case of negligible communication overhead, server-initiative algorithms tended to outperform source-initiative algorithms with the same level of information.
- In particular, the cyclic service algorithm performed surprisingly well in this situation, considering how little information it required.
- When communication overhead is significant, some algorithms, such as JSQ, may become much more expensive, depending on various factors such as the ratio of N to K.

Leer - Vide - Empty

Part II

The DMW Scheduler: Design and Implementation

Leer - Vide - Empty

Chapter 4

DMW: Policy

> In order to offer rational criteria for a right moral decision, the theories mentioned above take account of the intention and "consequences" of human action.
>
> John Paul II, *Veritatis Splendor*

This chapter presents the *scheduling policies* developed for DMW, starting with an outline of the properties that an ideal scheduler should exhibit.

The initial implementation of DMW was fully distributed, as shown in Figure 4.1. I shall discuss this policy and then explain why it proved not entirely satisfactory.

To overcome the deficiencies of the distributed scheduler, DMW was revised to employ a centrally arbitrated scheduling policy as shown in Figure 4.2, which was then, based on further experiments and user feedback, refined into its current form.

To conclude, I shall discuss the remaining open issues in the scheduler and how to address them.

When discussing scheduling policies, I will alternate between a discussion of scheduling in the entire network, scheduling from the point of view of a single user submitting jobs, and scheduling from the point of view of the operating system on a single machine. I shall use the terms *scheduler*, *individual scheduler*, and *local scheduler*, respectively, for the software responsible for implementing these three aspects of scheduling.

4.1 An Ideal Scheduler

Given a set of schedules S, each submitted by one of the individual schedulers and consisting of:

- A set of hosts

$$\mathcal{H} = \{h_1, h_2, \ldots, m_n\},$$

- A set of jobs

$$\mathcal{J} = \{j_1, j_2, \ldots, j_m\}$$

with each job j_i permitted to run on a subset of hosts $\mathcal{H}_i \subset \mathcal{H}$,

- A set of data dependences

$$\mathcal{D}_d = \{j_p \delta^d j_t \mid j_p, j_t \in \mathcal{J}\},$$

- A set of priority dependences

$$\mathcal{D}_p = \{j_p \delta^p j_t \mid j_p, j_t \in \mathcal{J}\},$$

assign the jobs to the hosts such that the following requirements are met:

Data Dependences $\forall j_p \delta^d j_t \in \mathcal{D}_d$: j_t does not get started until j_p has completed without errors.

Priority Dependences $\forall j_p \delta^p j_t \in \mathcal{D}_p$: j_t does not get started until j_p has been started.

Host Constraints $\forall j_i \in \mathcal{J}$: j_i does not get started on any host that is not a member of $\mathcal{H}_i$

Completion Guarantee Provided that $\mathcal{D} = \mathcal{D}_d \cup \mathcal{D}_p$ is acyclic, i.e.,

$$\forall j_i \in \mathcal{J} : \neg(j_i \delta^* j_i),$$

with

$$\begin{aligned} j_p \delta^* j_t &= j_p \delta j_t \vee (\exists q : j_p \delta j_q \wedge j_q \delta^* j_t), \\ j_p \delta j_t &= (j_p \delta^d j_t \in \mathcal{D}_d) \vee (j_p \delta^p j_t \in \mathcal{D}_p), \end{aligned}$$

and that no errors occur in the execution of any of the $j \in \mathcal{J}$, all $j_i \in \mathcal{J}$ eventually get executed.

In addition to these requirements (which clearly are non-negotiable properties of a correct scheduler), there are a number of further desirable properties:

Fairness If multiple users submit comparable schedules with similar jobs, dependence structures, and host sets to the scheduler simultaneously, they should receive a similar quality of service.

Interactive Transparency None of the hosts in $\mathcal{H}$ are ever subjected by the scheduler to resource demands adversely affecting the computing use of interactive users.

Progress Guarantee Regardless of load conditions, one job per individual scheduler is always kept running.

Performance Optimization Subject to the above constraints, jobs are executed as quickly as possible.

However, with the limitations of the environment and of our mechanisms, not all of these properties can be satisfied fully. In the next few sections, I shall discuss the major difficulties that an implementation faces in meeting these requirements.

4.1.1 Interactive Transparency

All operating systems that DMW supports have been designed for multiuser operation and shield users from each other to ensure that scheduling background jobs does not affect the reliability of the host or provide unauthorized access to the interactive user's data. However, background jobs may become noticeable (and thus disliked) if they decrease the performance of the host too much. There are several resources which are potential causes of contention between background and interactive processes:

CPU Availability

If the number of runnable processes on a host exceeds the number of CPUs, the operating system has to allocate each CPU to a process for a period of time (called a *time slice*) and preempt the process and schedule another process once the time slice expires.

The UNIX scheduler on each host handling local scheduling ([MBKQ96, Section 4.4] and [GC94, Section 4.6] discuss the algorithms for BSD 4.4 and System V Release 4, respectively) is usually quite capable of absorbing excessive CPU demands by background jobs without adverse effects on perceived performance for interactive jobs: UNIX scheduling favors interactive processes, i.e., processes that spend most of their time waiting on an external event and rarely use up an entire time slice.

In addition, DMW assigns a high *nice value* to the jobs it starts, directing the local scheduler to give these jobs a lower priority than jobs started by interactive users. As shown by the measurements in section 7.5, this keeps a CPU-bound job to about 15% of the CPU if a CPU-bound job started by an interactive user is also present.

Therefore, background jobs do not interfere with the CPU demands of interactive jobs. They do slow down computing intensive jobs started by interactive users somewhat, but this is usually considered tolerable.

Physical Memory

Another resource which foreground and background processes compete for is physical memory. Operating systems employ demand-paged virtual memory mechanisms [Tan87, sections 4.3–4.5], allocating physical memory pages for some fraction of the virtual memory pages that each process requests. Most processes exhibit some locality of reference and only access a relatively small fraction of their entire memory space (called their *working set* [Den68b]) during any phase of execution, and as long as the total size of the working sets of all processes running on a host does not exceed the available physical memory, the *page fault* rate of the host remains moderate.

As soon as available memory is exceeded, however, bringing in memory pages from disk requires expelling pages which are still part of the working set for some process and thus are likely to be needed back in memory soon, causing another page fault. The page fault rate increases sharply, and performance of the host drops as processes spend increasing amounts of time waiting on page faults, a phenomenon known as *thrashing* [Den68a].

While UNIX schedulers can detect thrashing and typically react to it by swapping out entire jobs to disk for a few minutes, thrashing remains a noticeable phenomenon and is highly disruptive to interactive users and detrimental to the performance of background jobs. Thus, the distributed scheduler should avoid placing jobs whose working sets are likely to exhaust the physical memory on the host.

Network Capacity

Processes theoretically also compete for network capacity. UNIX schedulers take only the most rudimentary precautions against attempts to inject an excessive amount of traffic into a network connection: Wright and Stevens [WS95] explain that the BSD networking implementation simply discards packets if the queues of unsent packets for a network interface grow too long.

In practice, however, with current local area network capacities and workstation speeds, the network traffic demands of background jobs do not affect the operation of regular jobs.

One situation in which network capacity does become a critical resource is if some of the hosts are connected by *low capacity links*, e.g. dial-up connections. Jobs running on these hosts can saturate these links with their NFS and X window system network traffic.

In our target environments, we do not expect such network topologies to affect many of the hosts, so we proceed on the assumption that such hosts are not listed as scheduling targets. Clearly, a scheduler operating on a fundamentally inhomogeneous network (e.g., two corporate intranets connected to each other by a slow link), would have to proceed on rather different assumptions, for instance abandoning the assumption of a common file namespace.

Progress Guarantee

The progress guarantee mentioned above may further interfere with the computing needs of interactive users. However, it exists because we were concerned about scheduler users suspecting a malfunction of the scheduler (and possibly switching to manual placement of jobs) if the scheduler refused to run any jobs at all at times of high load. The progress guarantee under these conditions simulates the behavior of a user sequentially executing the jobs in the schedule.

4.1.2 Performance Optimization

The above goal of "Performance Optimization" is rather vague, as there is a variety of plausible performance metrics which could be optimized. Krueger and Livny [KL87] compare the effects of various policies on:

- The mean *wait time* $\overline{\mathrm{WT}}$ of jobs.
- The mean *wait ratio* $\overline{\mathrm{WR}}$, where WR is defined as the ratio of wait time WT to process service demand X.
- The standard deviation of the wait time σ_{WT}.
- The standard deviation of the wait ratio σ_{WR}.

and observe that optimizations of some of these performance metrics is mutually exclusive. For instance, a first-come–first-served (FCFS) scheduling policy will minimize σ_{WT} since it does not discriminate in favor of shorter jobs, but, for the same reason, $\overline{\mathrm{WR}}$ and σ_{WR} for such a policy will tend toward infinity.

Since in our application, users are concerned with the execution time of an entire simulation, consisting of a mix of shorter and longer jobs, they are more likely to be concerned about $\overline{\mathrm{WT}}$ than about $\overline{\mathrm{WR}}$. Therefore, no attempt is currently made to minimize WR, especially since such an effort would require a job preemption mechanism.

4.2 The Distributed (`RStat`) Scheduler

The distributed implementation of the scheduler[1] was done under the assumption that a fully distributed architecture would work best and that a central coordinating instance would have disadvantages in performance and reliability. Due to external time constraints on the release of an implementation, we restricted the generality of host subsets, such that each job is either permitted on all hosts ($\mathcal{H}_i = \mathcal{H}$) or permitted on a single host only ($\mathcal{H}_i = \{h_k\}$).

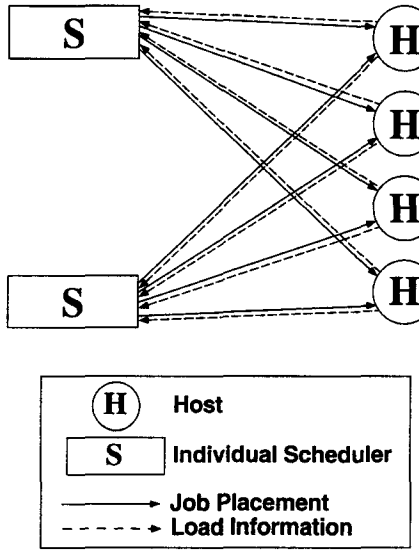


Figure 4.1: Distributed Scheduling

4.2.1 Distributed Scheduling Policy

The distributed scheduler is built around a rescheduling procedure that needs to be called at regular intervals (typically at least once a second). This procedure works as follows:

- The subset $\mathcal{J}_{\text{eligible}}$ of jobs not yet started which meet all of their data dependence and priority constraints is calculated.
- Many of the jobs in the schedule are very simple and merely have to copy or rename some files for the next simulator. In `DMW`, these jobs are written in Tcl (Tool Command Language [Ous94]) and executed by the individual scheduler

[1] Also named the `RStat` scheduler for its reliance on the `rstat` mechanism for obtaining host information

itself as soon as they are runnable. Up to $n_{\text{Tcl}} = 4$ (a somewhat arbitrary bound to limit the time spent in a single call to the rescheduling procedure) Tcl jobs are executed and, since they always finish immediately, the set of eligible jobs is then recomputed.

- Load averages for all the relevant hosts are then retrieved and weighted. There is no single obvious way to compare load averages on different platforms with each other. DMW uses the simple approach of dividing the reported load average λ^{rep} by a host specific weighting factor w_h to give a weighted load average

$$\lambda_w = \lambda^{\text{rep}} / w_h$$

  where w_h is set by the customer in a *host database*, a file listing for each eligible host:

  - The name or IP address.
  - The hardware platform.
  - The operating system version.
  - The w_h value.

- If any of the jobs are permitted on a single host only and jobs are already running at the moment, the first of these eligible jobs whose host has a normalized load average λ_w less than $\lambda_{\text{thresh}} = 0.5$ is started. If no jobs are running at the moment, the first eligible job is started, regardless of the load average.

- If any of the jobs are permitted to run on all hosts, the procedure repeatedly determines the host with the lowest load average and starts an eligible job on it until one of the following conditions is met:

  - No eligible jobs remain.
  - $n_{\text{Regular}} = 8$ jobs are running.
  - At least one job is running and none of the remaining hosts have load averages less than λ_{thresh}.

- Any hosts on which the two preceding steps have started a job are *embargoed*, i.e., excluded from being considered for further jobs, until either one of the jobs started on that host has terminated or 90 seconds have elapsed since the start of the last job on that host.

4.3 Evaluation of the Distributed Scheduler

Comparing the distributed scheduler with the objectives developed in section 4.1, we see that it fully enforces data and priority dependences, and fulfills the completion guarantee. Host constraints are enforced, but, as noted above, $\mathcal{H}_i$ is restricted to contain either a single host or to be equal to $\mathcal{H}$.

Furthermore, the progress guarantee is also fulfilled. However, the distributed scheduler makes no effort at fairness: Available hosts are, at best, allocated to individual schedulers on a first come, first served basis, and in fact, as discussed in section 4.3.2 below, often overallocated. Furthermore, the distributed scheduler is also

rather limited in its efforts at interactive transparency and good performance because of its inadequate information policy and because it does not address the problem of host contention.

4.3.1 Inadequate Information Policy

The only host resource information that `rstat` provides is the load average, which only gives an indication of CPU utilization. Not only is there no information about memory utilization available, but, absent that information, even the load average may become entirely misleading in some situations:

When a host has overcommitted its physical memory and starts thrashing, it spends an increasing percentage of its time waiting for virtual memory page transfers to complete and a decreasing percentage doing actual computation. Some operating systems, e.g., SunOS 4.x, do not count processes waiting for virtual memory transfers as runnable and thus will report a decreasing load average, inviting the scheduler to send even more jobs to that host. Other operating systems, e.g. SunOS 5.x, count those processes as runnable when calculating their load average, and thus do not suffer from this problem.

4.3.2 Host Contention

As Theimer and Lantz pointed out (Section 3.3.2), if the individual schedulers in a distributed system all base their independent scheduling decisions on the same information, it frequently happens that several schedulers attempt to start a job on the same host.

This risk is worsened by the fact that the load average reported by `rstat` does not change very quickly and the impact of a new job started on the host is not reflected fully in the load averages in the beginning of the job's existence. Individual schedulers avoid that effect through the 90 second embargo policy described above, but they have no information about jobs started by other schedulers.

Theimer and Lantz propose to annotate job scheduling requests with the load on which the scheduling decision was based, and to reschedule the job if the current load on the host significantly exceeds that recorded load. However, it is doubtful that this approach would work with our distributed scheduler: In our system, job transfer times are very short, so the above reporting delay is considerably longer than the typical interval between the scheduling decision and the start of the job.

4.4 The Arbitrated (`UMol`) Scheduler

The arbitrated implementation of the scheduler[2] intends to permit arbitrary sets of permissible hosts for each job and to address the performance and fairness limits of the distributed scheduler.

For the latter reason, the arbitrated scheduler introduces a central coordinator similar to the GLOBAL or CENTRAL schedulers studied by Zhou and Ferrari 3.3.1. However, our central coordinator does not merely collect and pass on load information, as the GLOBAL coordinator does, nor does it actually place the jobs, as the CENTRAL coordinator does. Instead, it collects the load information and information about the

[2]Also named the `UMol` scheduler for relying on the `DMWUmpire` arbitration daemon and the `DMWMole` resource measurement daemon for its operation

jobs, decides where the jobs should be placed, and tells the individual schedulers where to place the jobs. To emphasize this decision making role and the concern with fairness, I call the central coordinator the *arbiter* or *umpire*.

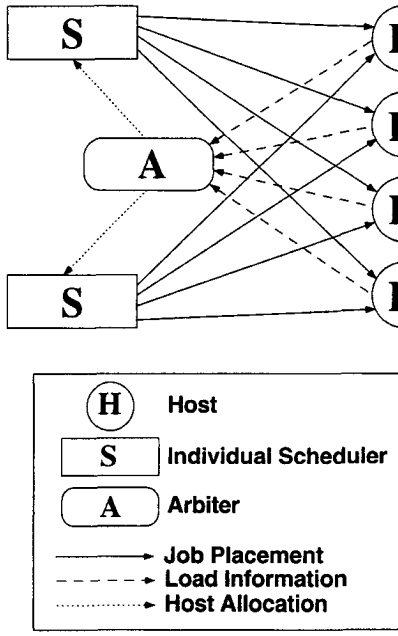


Figure 4.2: Centrally Arbitrated Scheduling

4.4.1 Distributed Rescheduling Policy

Policy in the arbitrated scheduler is shared between the rescheduling procedure in each of the individual schedulers and a central `DMWUmpire` daemon. The rescheduling procedure is essentially a considerably simplified version of its counterpart in the distributed scheduler:

- The set of eligible jobs $\mathcal{J}_{\text{eligible}}$ is calculated.
- The first n_{Tcl} Tcl jobs are executed and $\mathcal{J}_{\text{eligible}}$ is recomputed if necessary.
- The set of useful hosts is calculated as

$$\mathcal{H}^0_{\text{useful}} = \bigcup_{j_i \in \mathcal{J}_{\text{eligible}}} \mathcal{H}_i$$

- $\mathcal{H}^n_{\text{useful}}$ is then transmitted to the `DMWUmpire` and a request is sent to grant permission to run a job on one of the hosts in the set.
- If the `DMWUmpire` responds with a suitable host, the first job j_k able to run on that host is started. Otherwise, the rescheduling procedure quits with a status signalling the scheduler to retry in a few seconds.
- $\mathcal{H}_{\text{useful}}$ is then recomputed as $\mathcal{H}^{n+1}_{\text{useful}} = \mathcal{H}^n_{\text{useful}} \setminus \mathcal{H}_k$, which is, of course, only a subset of the true value, but faster to compute.
- As long as the resulting $\mathcal{H}^{n+1}_{\text{useful}}$ is not empty, it is retransmitted to the `DMWUmpire` and another host is requested.

4.4.2 Centralized Arbitration Policy

The above distributed policy is combined with the global arbitration policy in the `DMWUmpire` daemon. Since host requests from the individual schedulers reach the arbitration daemon asynchronously and may not even get handled exactly in the order in which they arrive, `DMWUmpire` separates the task of allocating hosts into two phases:

- In the *apportioning* phase, hosts are distributed globally by reserving them for individual schedulers based on the stated interests of the schedulers and on their priority.
- In the *granting* phase, individual schedulers confirm their reservations for the hosts assigned to them.

The next two sections examine these phases in more detail.

Apportioning

For each of the individual schedulers s, `DMWUmpire` keeps track of

$\mathcal{H}_s$ the $\mathcal{H}_{\text{useful}}$ set announced by that scheduler,

n_s the number of jobs it has running,

t_s the last time a host was granted to the scheduler,

h_s the host, if any, currently apportioned to the scheduler.

When a round of apportioning is initiated:

- The individual schedulers are sorted by n_s and t_s. A scheduler i has a higher priority than a scheduler j if $\langle n_i, t_i, i\rangle < \langle n_j, t_j, j\rangle$.
- Apportioned hosts h_s not requested by their scheduler after 90 seconds are taken away from the schedulers again. Since hosts are only apportioned to schedulers who have previously indicated their interest in them, this should happen only when a scheduler crashes (Occasionally, a scheduler may announce a more restrictive $\mathcal{H}_s$ which no longer includes the host apportioned to it. `DMWUmpire` checks for that case when a $\mathcal{H}_s$ gets updated and frees h_s if $h_s \notin \mathcal{H}_s$).

- Unapportioned hosts are sorted by their score σ, a weighted assessment of their weighted load average λ_w (as defined in section 4.2.1) and free memory μ (in kilobytes):

$$\begin{aligned}
\sigma &= \sigma_\mu + \sigma_\lambda \\
\sigma_\mu &= \begin{cases} 120 & \text{if } \mu > 200000 \\ \mu/50 & \text{if } \mu < 5000 \\ 100 + \mu/10000 & \text{otherwise} \end{cases} \\
\sigma_\lambda &= \begin{cases} 0 & \text{if } \lambda_w > 4.9 \\ 121 - 35\lambda_w & \text{if } \lambda_w < 0.6 \\ 112 - 20\lambda_w & \text{otherwise} \end{cases}
\end{aligned}$$

  These somewhat speculatively chosen scoring functions for σ_μ and σ_λ score memory availability and load each on a scale from $0 \ldots 120$ where 120 implies a perfect score, 100 a satisfactory score, and 0 a hopelessly overloaded host.

- The algorithm then iterates through hosts with $\lambda_w < \lambda_{\text{thresh}} = 0.6$ and $\mu > \mu_{\text{thresh}} = 5000$. Each of these hosts is apportioned to the highest priority scheduler for which it is useful.

- Finally, the algorithm iterates through schedulers with no jobs running and no host apportioned to them. Each scheduler is apportioned the least loaded host useful for it.

Granting

When an individual scheduler requests a host, the following algorithm is executed:

- If no host has been apportioned to that scheduler, the apportioning algorithm in the previous section is executed.

- If a host h_i has been apportioned to that scheduler (as a result of the previous step or of an earlier apportioning), permission is granted to the scheduler to start a job on this host, and h_i is embargoed as described above for the distributed scheduler, i.e., excluded from scheduling for the next 90 seconds or until the next time a job scheduled on it terminates.

- If the apportioning failed to apportion a host to the scheduler, the request is denied, with the intent that the scheduler should wait a few seconds and then reapply for a host.

4.5 Evaluation of the Arbitrated Scheduler

The arbitrated scheduler fully supports data and priority dependences, enforces host constraints (with the $\mathcal{H}_i$ allowed to be arbitrary subsets of $\mathcal{H}$) and guarantees completion and scheduling progress as defined in section 4.1. The arbitrated scheduler provides fairness through its central arbitration and improves interactive transparency and performance by addressing two of the key weaknesses of the distributed scheduler:

- Since both load averages and memory utilization data are available to the scheduler, it no longer risks considering a system lightly loaded when it is in fact thrashing.

- Since hosts are apportioned centrally, there is no longer any host contention among the individual schedulers.

However, the arbitrated scheduler is not entirely unproblematic, either. On one hand, the introduction of a centralized component into any distributed system raises some fundamental concerns about the performance limitations and robustness of the process implementing this central function. In section 4.5.1, I address those concerns. On the other hand, practical experience has shown that the arbitrated scheduler is not as successful at preventing overloaded hosts as it was hoped to be. In particular, there are difficulties related to the inability to adequately predict the future resource demands of jobs and the future availability of resources on a host; I shall discuss these issues in section 4.5.2. Furthermore, I shall argue in section 4.5.3 that the scoring functions used above do not reflect the expected performance of a job on a given host very well, and I shall present an improved load index.

4.5.1 Performance and Robustness Concerns

As mentioned in the discussion of the research of Theimer and Lantz (section 3.3.2) and Shivaratri et al. [SKS92] (section 3.3.3), advocates of a fully distributed approach raise primarily two fundamental objections against a central coordinator in a scheduling system:

- Will the central coordinator become a performance bottleneck as the number of hosts in the cluster increases?

- What happens if the central coordinator, or the machine on which it runs, fail?

I believe that both of these objections can be answered convincingly.

As practical experience shows, *performance* concerns are rarely justified. The central coordinator in Condor is currently [LBRT97] able to coordinate the activities of more than 300 hosts without any performance problems. While `DMW` has never been tested with that many hosts, our experience so far confirms that resource usage of the central coordinator is negligible.

Moreover, Theimer and Lantz point out out that a centralized algorithm acquiring the same amount of information about a system requires much less networking and performance measurement resources than a comparable distributed algorithm, so it will be *more* scalable than the distributed algorithm.

There are well known algorithms to detect and handle the *failure* of the central coordinator. Detection is usually accomplished by defining a sufficiently long timeout and declaring the coordinator failed if it has not handled a request within that time span. If only the central coordinator has failed (e.g., due to a programming error), it can simply be restarted.

However, if the host itself on which the central coordinator ran has failed (e.g. due to a hardware failure or a network disruption), this solution is not applicable. In some systems, the individual schedulers choose a new host for the central coordinator through a *distributed election*. Algorithms for this are well-known, but somewhat complex.

DMW uses a different approach to this problem: Execution of schedules stops, the schedules must be restarted when the host is running again or, if the failure is more than intermittent, a different coordinator host must be configured in the system. This simple strategy is justified by the environment in which DMW runs:

The common file namespace required by DMW requires a global file system, which in a typical environment will reside on a single NFS server. This server is already a single point of failure; if it is designated as a coordinator host, the scheduler has nothing to gain from any sophisticated recovery strategy: While it would be possible to elect a new coordinator host when the NFS server fails, it would still not be able to carry out any simulations since the project file systems would not be available. Conversely, since the central file server is crucial to all work in the workstation cluster anyway, it is likely to be designed more robustly and maintained with more care than an average host.

4.5.2 Inability to Predict Future Resource Conflicts

While the arbitrated scheduler's handling of immediate placement decisions is mostly successful for shorter running simulations, it may lead to disaster with long running simulations. Three typical scenarios for such long term problems are:

Multiple Job Thrashing A big simulation is started on a host. After the 90 second embargo has elapsed, the scheduler still finds the host attractive for further jobs and starts a second big simulation. The two simulations run well for several hours until their combined working sets exceed the physical memory of the host, and then the jobs starts to thrash. While the operating system may be able to reduce the impact of the thrashing somewhat by alternately swapping one of the simulations entirely out of memory, performance suffers.

Single Job Thrashing A single big simulation is started on a host. Although no further jobs are started on that host, the simulation is so big that eventually its working set exceeds physical memory. Again, thrashing ensues, but since it is caused by a single job, process swapping cannot improve the situation in this case.

Interactive Interference A big simulation is started on an idle host. After some time, an interactive user starts using the host and finds that the background process interferes with his work. Although the scheduler tries to avoid placing too high a load on a host, some activities, such as audio output to the built-in speaker of a workstation, are extremely sensitive to any background activity [Zim98].

All of these scenarios develop into unsatisfactory situations with both the distributed and the arbitrated scheduler because of three fundamental assumptions in the scheduling policies:

Unpredictable Resource Demands No information is available about the resource demands of jobs at the time they are placed.

Unpredictable Resource Availability It is impossible to predict future resource availability, as a user may wish to start using a host interactively and then restrict resources available to background jobs.

Non-preemption of Jobs Once a job has been placed on a host, it runs to completion on that host.

Clearly, it is impossible to avoid situations where resource demands on a host exceed availability if those three assumptions are maintained simultaneously. There are, however, various possible solutions based on abandoning or modifying some of these assumptions.

Dedicated Hosts

One solution to the problem of interactive interference on a host is to ban interactive users from that host and set up a *dedicated cluster*. However, that solution, although simple to implement from a software point of view, is not very attractive as it usually requires additional hardware, and as it does not address the problems of job thrashing.

Job Termination

Another solution to jobs that exceed available resources is to simply *kill* them, an approach that is used, e.g., in Condor [LBRT97] when other methods are not practicable for some reason. The scheduler has then some additional information (the process size at the time the job was killed and the interactive demand on the host) to guide it toward restarting the job on a machine more suited for a successful execution.

Job termination is simple to implement and can address all of the problems described above. However, termination has several drawbacks:

- On slow hosts with small amounts of physical memory, a majority of jobs scheduled might eventually get terminated.
- Since simulation jobs typical for our work load have a tendency to grow gradually, termination will occur only after a considerable amount of processing has been performed already.
- Frequent termination may cause both scheduler users and foreground users to question the correct operation of the scheduler.

Thus, job termination should only be used as a final resort, and coupled with a permanent feedback mechanism (annotating the observed resource usage of the job) to prevent future attempts to schedule the same job on the same machine again if thrashing due to a static resource mismatch, rather than interactive interference, was the reason for termination.

Job Suspension

Instead of killing a job, it is also possible to *suspend* a job by sending it a `SIGSTOP` signal (the programmatic equivalent to typing `Control-Z` to suspend an interactive UNIX process). Job suspension can be done statically (background jobs on certain hosts are suspended during working hours), dynamically (background jobs are suspended when interactive activity is detected), or upon request by interactive users. Sometimes suspension is also combined with other techniques, e.g., in Condor, where jobs are suspended upon detecting interactive activity and later resumed if the interactive activity ceases again within a few minutes and otherwise migrated or terminated.

Job suspension is simple to implement and works well against interactive interference and multiple job thrashing (since the operating system will swap out suspended processes if necessary), but is not effective against single job thrashing.

Global Job Limits

One solution available to the arbitrated scheduler thanks to its centralized component is to track the number of background jobs running simultaneously on a host and enforce a *global limit*, i.e., a maximal number of jobs placed on it by all individual schedulers combined.

Originally, no such limits were enforced because of the assumption that multiple simultaneous jobs would be able to overlap computation and I/O and thus complete faster in parallel than if run serially. However, practical experience shows that I/O plays a negligible role in longer running simulation jobs, and that they are almost totally CPU bound (see section 7.4.1). Thus, as soon as the number of jobs on a host exceeds the number of processors, the increased risk of thrashing far outweighs the potential benefits of execution overlap.

Therefore, the arbitrated scheduler was modified to enforce job limits specifiable in the host database and usually set equal to the number of processors on a host. By lowering resource usage, job limits have the potential to mitigate all of the problems mentioned above, although they do not provide a perfect solution to any of them.

User Hints

As opposed to the assumption of unpredictable resource demands outlined above, scheduler users usually have a reasonably accurate estimate of the resource demands of their simulation jobs and often are able to refine this estimate further while working with a simulation. The scheduler could thus be modified to base its choice of $\mathcal{H}_i$ for a job on a *user provided hint* for the size of the job (large, medium, small) as well as the program executing the job.

User hints are easy to implement and, when employed by a competent user and combined with global job limits, can eliminate thrashing entirely. On the other hand, they are not entirely effective against interactive interference.

Checkpointing and Job Migration

The most powerful solution to the above problems, but also the one requiring the most difficult implementation, is *job migration*, a facility to save the state of a job in a *checkpoint*, terminate the job, and restart it on a different host.

Although a job migration mechanism for `DMW` was designed and, to some extent, implemented, it was never integrated into `DMW`. Section 5.7 discusses the problems of the implemented migration mechanism and potential alternatives.

4.5.3 Inadequate Performance Index

A third shortcoming in the scheduling policy is the unsatisfactory way in which the performance index is computed from the load average. In particular, the formulae for both the distributed and the arbitrated scheduler compute the same load index values for all hosts with load averages of 0, regardless of the hardware speed of the host.

This has the consequence that differences in installed memory and small fluctuations in the load average tend to have a much bigger influence on the ordering of the hosts than hardware performance. In our research environment, this rarely leads the scheduler to make bad choices, since the physical memory installed in hosts happens to correspond quite closely to their computing performance and thus contributes to a

reasonable load index, but such a relationship does not, of course, exist in every workstation cluster.

The somewhat arbitrary original load index should therefore be replaced with an index designed to give a reasonably accurate performance estimate, at least for the short term future of a host, from its current load average λ and its basic performance β. Since the measurements in chapter 7 show that longer running simulation jobs are almost exclusively CPU bound, β can be defined as the number of executions of a CPU bound reference workload per time unit if that workload is the only job on the host ($\lambda = 1$).

Assuming that CPU time is divided equally between all jobs ready to run (which is only strictly true if all nice values are the same), and that the new job to be started will add 1 to the existing load average (which is accurate for a purely CPU bound job), the work that an uniprocessor host could perform for the new job then becomes

$$\sigma_\lambda = \beta \frac{1}{\lambda + 1}.$$

This approach is easily extended to a host with n processors with the caveat that running a single job on a multiprocessor host does not, of course, speed up its execution beyond the speed of a single processor, so simply multiplying the above value by n would significantly overestimate the performance of the host at low loads. Instead, performance can be estimated as

$$\sigma_\lambda = \beta \frac{n}{\max(n, \lambda + 1)}.$$

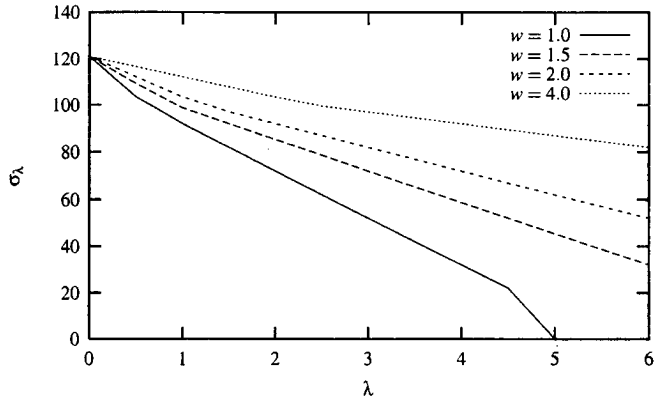
The threshold for job placement on hosts can be defined by choosing the least powerful host still to be considered interesting and the λ at which it still should receive jobs and defining λ_{thresh} as the σ_λ of that host under these conditions.

Figure 4.3 shows the differences between the old and new performance indices: In the range of $0 \leq \lambda \leq 1$, which is the most important for job placement, the old index is least sensitive to w, while the new index is most sensitive to β in that range. Furthermore, for multiprocessor machines with $n > 1$, performance is correctly predicted to remain constant until the load average reaches the number of processors, while the linear performance prediction of the old curve would either overestimate performance at low loads or underestimate it at higher loads.

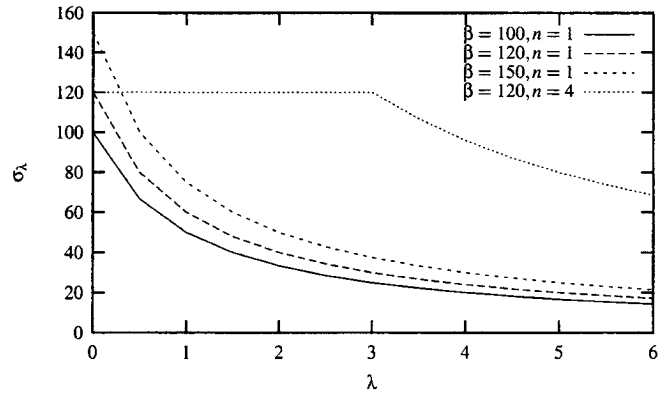
On the other hand, by representing performance differences between machines more accurately, the new load index formula, applied in isolation, bears the risk of having the scheduler start too many jobs on fast machines: If, as an example, λ_{thresh} in our environment is defined such that an unloaded 50MHz Sun SPARCstation 10 is just acceptable to the scheduler for job placement, the scheduler would consequently consider the fastest server available (a Sun Ultra2-2300) suitable for further jobs up to a load average of 3.5. Placing so many jobs on a single processor machine is at best useless (as they would have completed just as fast if run sequentially), and often detrimental to performance if the jobs grow to exhaust physical memory on the machine.

Therefore, it is essential for the operation of the new load index that it be combined with global job limits as described above in section 4.5.2. Furthermore, as the measurements show, some of the other parameters originally used were not well chosen:

- μ_{thresh} is too low (the amount of memory guaranteed is massively exceeded even in simple simulations) and should be increased to at least 20000.
- The embargo period of 90 seconds is not long enough for the load average to reflect the performance impact of a new job and should be increased.



(a) Original: $\sigma_\lambda = \begin{cases} 0 & \text{if } \lambda_w > 4.9 \\ 121 - 35\lambda_w & \text{if } \lambda_w < 0.6 \\ 112 - 20\lambda_w & \text{otherwise} \end{cases}$



(b) New: $\sigma_\lambda = \beta \frac{n}{\max(n, \lambda+1)}$.

Figure 4.3: Original and new load indices for DMW

Chapter 5

DMW: Mechanism

> Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and weigh only 1 1/2 tons.
> Popular Mechanics, *March 1949*

This chapter discusses the basic mechanisms underlying the DMW scheduler: The remote execution and job control facility in section 5.2, the remote monitoring daemons in sections 5.3 and 5.4, the central arbiter in section 5.5, and the job migration mechanism in section 5.6ff. All of these components share a common security concept, which I shall describe first.

5.1 Security

Since DMW is implemented with common internet protocols on standard hardware which is usually attached to enterprise wide intranets and often to the Internet, some attention has to be paid to security aspects. Cheswick and Bellovin [CB94] categorize security risks into:

Stealing Passwords by gaining access to the password file via network programs (e.g., guest logins, `ftp`, `tftp`) or by capturing network traffic (a practice also referred to as *snooping*) and searching it for cleartext passwords transferred e.g. as part of `telnet` or `ftp` session.

Social Engineering and Information Leakage i.e., gaining access or acquiring information by approaching authorized personnel or by reading publicly available information (company telephone books).

Bugs and Backdoors in network software, e.g. undocumented "debugging modes" or unchecked read operations into limited buffers.

Authentication and Protocol Failures when authentication mechanisms are either inherently weak or are defeated, e.g., by PC clients creating unverified user names.

Denial of Service attacks that intend to disable ordinary operation of hardware and software rather than gain unauthorized access.

Therefore, it is essential for DMW not to introduce any additional weaknesses in any of these areas. A number of different techniques are employed to attain that goal:

- All daemons—the monitoring daemon, the arbitration daemon, and the job control daemon—perform only a small, restrictively defined set of actions. In particular, none of the daemons will start a new job on behalf of a different user than the one who started them, effectively delegating authentication for these critical actions to the UNIX remote shell (`rsh`) system.

- By using `rsh`, DMW also avoids the exchange of cleartext passwords over network connections.

- All daemons operate at the lowest privilege level necessary to do their jobs. The monitoring daemon, which on most architectures needs superuser permissions to be able to access kernel statistics, compensates by having a highly restricted command set.

- All daemons bind themselves to *ephemeral TCP ports* [Ste94] whose port number is chosen arbitrarily by the operating system. Additionally, they require that a 16-bit numerical password be given on connecting. To get the port number and password, a client has to start a process on the host, again delegating authentication to `rsh`.

- The monitoring daemon and the arbitration daemon guard against denial-of-service attacks by allowing a very large number of connections to be active at any time. All daemons detect invalid passwords and failure to provide a password in a timely manner. The authentication protocol is implemented entirely without blocking I/O operations.

- All I/O operations guard against buffer overruns.

Nevertheless, the described system has some security flaws that are potentially exploitable by determined attackers:

- The 16 bit connection password is rather short and thus subject to *brute force* attacks, especially since in practice, ephemeral TCP port numbers in most TCP/IP implementations are not assigned randomly, but in a more or less sequential fashion. This weakness could be remedied by choosing longer passwords.

- DMW forces the use of `rsh` and its associated system of *trusted hosts*. It has been shown in recent years that this system is susceptible to attacks such as IP *spoofing* (hosts changing the source IP address of packets they send to the address of a different host).

  The `ssh` *secure shell* is a more secure alternative to `rsh` and DMW could be changed with limited effort to support `ssh`. However, `ssh` is not in wide use yet and shipping it with DMW would add significant further complication to the software installation process.

- Even though no passwords are exchanged in any DMW related network connections, the simulator output data flowing through them is potentially sensitive information which could be picked up by packet snooping. `ssh`, which provides encrypted socket connections, would also offer a solution to this problem.

Thus, it is clear that a production system running DMW should not be exposed to the Internet without protection through strong firewall systems as described in [CB94].

However, against in-house attackers, who usually in a commercial enterprise do not have the undisturbed time and resources to mount sophisticated brute force password guessing, IP spoofing, or packet snooping attacks, the protection provided should be quite adequate.

5.2 Remote Execution and Job Control

This most basic part of the scheduler allows to:

- Start a new job on any host from any other host (including the ability to start a job on the same host).
- Receive data written on the job's standard output and error channels as it is generated.
- Automatically enable access to an X window server if needed.
- Check whether a job is still running.
- Send signals to a remote job to temporarily halt it, resume it, or terminate it prematurely.
- Save the connection parameters for a remote job to a file, close the connection to a job, and resume it later.

It would be impractical to build the features listed above into every program to be managed by the scheduler. Therefore, each job is controlled by an instance of a *job control daemon* named `DMWLaunchPad` which starts the actual job as a child process and controls it, as shown in Figure 5.1.

5.2.1 Startup Negotiation

At startup, `DMWLaunchPad` creates a listening socket for reconnection attempts and prints the port number of the socket and a numeric password in a greeting message. It then accepts a number of setup commands for

- Setting the name of the log files for the standard output and error streams of the job.
- Setting the directory in which the job is to be executed.

5.2.2 Starting the Job

Finally, `DMWLaunchPad` receives the command line to execute, including environment variables. It forks off a child process and gives it a new *process group* so it is possible to send signals to the process and all its subprocesses. If the command line is simple, without quote characters, shell wildcard characters or environment variables, `DMWLaunchPad` directly passes it to the `exec()` system call.

Usually, however, the command line needs wildcard expansion, quote processing, and environment variable manipulation, so `DMWLaunchPad` transforms it into a short Bourne shell (`/bin/sh`) script and starts a shell process to execute it. The script always ends with an exec command, so the shell process is immediately replaced by the job to be executed.

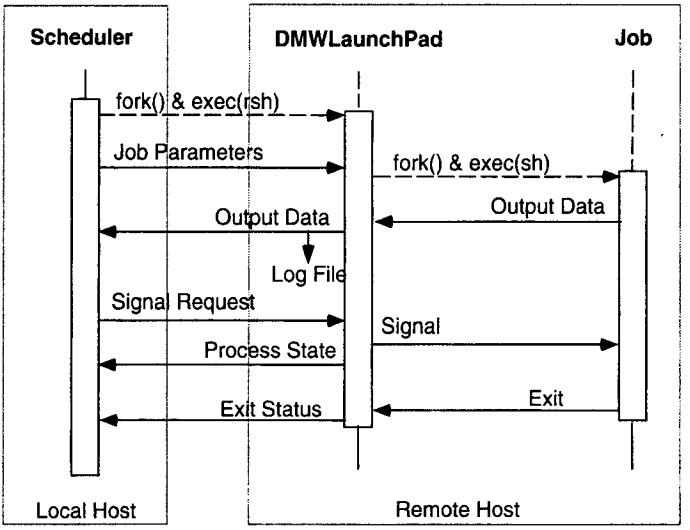


Figure 5.1: Operation of `DMWLaunchPad`

5.2.3 Job Monitoring

As soon as the job is running, `DMWLaunchPad` enters into a loop handling requests from the job and the scheduler.

- When the job writes data to its standard output or error streams, the data is copied to the corresponding log file and, if the scheduler has not closed its data sockets, also sent to the scheduler.
- When the scheduler requests that a signal be sent, `DMWLaunchPad` delivers it to the job process group.
- When the job terminates, `DMWLaunchPad` writes its exit status to the status file and quits itself.
- When a connection on the reconnection port arrives and completes the password handshake, `DMWLaunchPad` reconnects to a different scheduler instance.

5.2.4 Reconnecting

At startup, `DMWLaunchPad` prints the port number of the reconnection socket and the password. Based on this information, the scheduler can reconnect after it had been disconnected (e.g., because it ran on a *portable computer* which was temporarily taken off the net) by:

- Creating two new listening sockets for the standard output and standard error.

- Connecting to the reconnection socket.
- Sending to the socket the port numbers of the new sockets, the password, and whether any backlogged data is wanted.

If the password is correct and the handshake is completed within 60 seconds of connecting to the reconnection socket, DMWLaunchPad switches the standard streams to the new sockets. If a backlog was requested, DMWLaunchPad copies the requested amount of data from the log files to the new standard streams.

5.3 Remote Host Monitoring Using rstat()

Except for random transfers, every scheduling policy needs information about remote hosts, such as their

- Load average (number of runnable jobs, averaged over time)
- Memory utilization
- Network load

In the first implementation of DMW, this information was obtained using the standard rstat() call. Due to the limitations of this approach, the current implementation uses the custom written DMWMole daemon, discussed in section 5.4, instead.

5.3.1 Calling rstat()

In an initial implementation of the monitoring facility, the rstat() call was used to obtain load information from remote hosts. rstat() is a *Sun RPC* [Blo92, Mic88b] stub procedure which communicates with the rstat daemon (rpc.statd) on the remote host to return an array containing the average number of runnable jobs in the last 1, 5, and 15 minutes.

While neither Sun RPC nor rstat() are Internet Official Protocol Standards [JP94] or part of any of the POSIX standards, they are freely available and widely deployed due to the popularity of NFS [Mic88a].

The programming interface to rstat() is simple: The call

```
int rstat(char *host, struct statstime *statp)
```

fills in the statstime structure and returns a status code, analogous to stat(). The load averages are stored in the avenrun field of the structure. rstat() is implemented using the high level RPC call

```
callrpc(host, RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
        xdr_void, (char *)NULL, xdr_statstime, (char *)statp);
```

This call translates into a series of low level RPC calls, which use socket primitives to

- Pack the RSTAT program number, program version, and procedure number into a *port mapping* request and send it to UDP port 111 on the remote host.

- Wait for a response message containing the UDP port number for `rstat`.
- Send an UDP packet to that port. There is no input data to the `rstat()` call, so all the packet contains is RPC information like the program and procedure numbers.
- Wait for a response message and call the `xdr_statstime` XDR procedure to translate it from the system independent network representation to the system specific data representation.

Since the UDP protocol does not guarantee reliable delivery of packets, both the port mapping request and the rstat request are retried in 5 second intervals, up to 12 times for the port mapping request and 5 times for the rstat request.

5.3.2 Problems With `rstat()`

While `rstat()` was convenient for a prototype implementation, it turned out to have disadvantages in several respects:

Insufficient Information

`rstat()` only provides the load average of a host, which in the ISE simulation environment can be a quite misleading figure. As argued in section 4.3.1, a realistic information policy needs to include memory utilization figures, which `rstat()` is unable to provide.

Uncontrollable Performance

While `rstat()` has a convenient calling interface and performs well in the average case, an attempt to call rstat() for an inoperative host only fails after a long delay (as discussed in section 5.3.1, this delay will be at least 60 seconds) during which the process is blocked. While the timeout could be controlled by forgoing the high level interface and accessing the call through the low level RPC interface, doing so would sacrifice the simplicity of the `rstat()` monitoring implementation and thus defeat one of the main advantages of this version.

Security Concerns

While there are no known weaknesses in `rstat()` or its daemon that would allow an attacker to break into a host, the information provided by the protocol could be of potential use for an attack and the fact that the daemon hands out this information indiscriminately to any inquiring host must be considered *information leakage*. Consequently, some security scanning tools like SATAN [FV93] and ISS [Sys96] classify the presence of an `rstat` daemon on a host as a (low risk) security problem and some system administrators will disable the `rstat` daemon on their hosts.

Availability

Installing an `rstat` daemon on customer hosts where it is not yet installed usually requires superuser privileges, and, as discussed above, might be objectionable on security grounds.

5.4 Remote Host Monitoring Using DMWMole

Due to the problems discussed in the previous section, we decided to write a new monitoring facility for the centrally arbitrated scheduler: Target hosts run a daemon named DMWMole which, consistent with the security mechanism outlined in section 5.1, accepts TCP connections on an ephemeral port and responds to queries about the load average and memory utilization of the host. DMWMole maintains no internal state, so instances can be shut down at any time and will simply be restarted by the scheduler when needed.

5.4.1 Access to Statistical Information

The facilities providing access to performance statistics vary widely between different Unix kernels and are sometimes not well documented. Most kernels require the programs getting such information to have superuser privileges, so correct operation of the code is critical.

Fortunately, suitable source code was already available for all target platforms:

- For most UNIX platforms, the top program [LeF96] is available to give an interactive overview of load averages, memory utilization and the top CPU consuming processes on a host. top is widely ported, source code is available, and all platform dependent code is packaged in separate modules with an abstract interface. Those platform specific modules with a few modifications served as the basis for the platform dependent parts of DMWMole.

- When DMWMole was written, top was not available on IBM workstations running AIX[1]. Therefore, source code from the monitor program [Mak94], an AIX specific program with a superset of the functionality of top, was adapted to write a top style platform specific module providing the subset of kernel information needed by DMWMole.

5.4.2 Normalized Memory Statistics

The data provided by top represent the performance of one host with one particular platform. For these data to be useful to a heterogeneous scheduler, however, they have to be comparable to each other to some extent. Therefore, the raw figures reported by the kernel have to be adjusted in some cases to compensate for distortions introduced by variations in the *accounting policies* among the different operating systems.

DMWMole reports the total amount of installed memory and the amount of free memory. While the amount of free memory reported seemed comparable on most target platforms with a similar load, DMWMole on Solaris consistently reported a lower amount of free memory than other platforms. Apparently, the page replacement policy of Solaris tends not to reduce the working set [MBKQ96, Section 5.1] of inactive processes as long as there still is free memory in the system, while other UNIX kernels are more aggressive in reducing working sets.

Therefore, the platform specific DMWMole module for Solaris was adjusted to count 50% of the working set of inactive processes (defined as processes using less than 0.5% of the CPU) as free memory. This adjustment succeeded in making memory

[1] The current version of top (3.4), supports AIX

reports from Solaris hosts more consistent with other platforms and DMW no longer avoids assigning jobs to Solaris hosts.

A similar problem occurred on DEC Alpha workstations running OSF/1: Pages that are not referenced for some time by the process that owns them are marked as *inactive*, a state distinguished from *free*. Inactive pages are eligible for being reassigned by the kernel without further notice, but if the process references the page before it is reassigned, it becomes *active* again. The top host specific module for OSF/1 did not count inactive pages as free, which again underestimated free memory on OSF/1 hosts compared to other operating systems. Therefore, this module was adjusted for DMWMole to count 7/8 of the inactive pages as free memory.

5.4.3 Queries using DMWMole

To start monitoring a remote host, the scheduler[2] uses rsh to start up DMWMole using a hardcoded path. In the ISE distribution, this path refers to a Bourne shell (/bin/sh) script determining the platform of the host and selecting the right program binary to execute.

DMWMole returns a line consisting of

- A *version* identifier containing, in decimal form, the date DMWMole was last modified in a client visible way.
- The ephemeral TCP port chosen by the currently running instance of DMWMole as the listening port.
- The random password chosen by the currently running instance of DMWMole.

The scheduler then connects to the port and presents the password to authenticate itself. While this indirect technique has some performance overhead, it has the advantages of

- Restricting access to the DMWMole to clients with rsh access to the host, consistent with the security principles described in section 5.1.
- Avoiding the need to reserve a fixed port number for DMWMole.

Once the connection is established, the scheduler can query DMWMole using the one letter commands shown in table 5.1.

L	Return a line listing the non-normalized 1 minute, 5 minute, and 15 minute load average of the host.
M	Return a line listing total installed memory and the normalized amount of free memory on the host.
Q	Disconnect the scheduler from the DMWMole.
S	Shut down the DMWMole. This command is never generated by the scheduler, but can be sent manually.

Table 5.1: Commands accepted by the DMWMole daemon.

[2]This word is here used in the abstract sense denoting the entire system. Since DMWMole is part of the arbitrated scheduler, it will in fact only be contacted by the arbitration daemon, but it could in principle be used in schedulers with different information flow structures

5.5 Central Arbitration

As discussed in section 4.4, the arbitrated scheduler relies on the `DMWUmpire` arbitration daemon to make placement decisions. While `DMWUmpire` is constructed similarly to `DMWMole` in many respects, it maintains some internal state and has to be concerned with reconstructing it after a process failure.

5.5.1 State Maintenance and Reconstruction

As discussed in sections 4.4ff, the arbitration daemon bases its decisions on the following data:

- The $\mathcal{H}_s$ set of useful hosts announced by an individual scheduler,
- The number n_s of jobs it has running,
- The last time t_s a host was granted to the scheduler,
- The load average λ and available free memory μ of each host.
- In recent versions, the current number of jobs running on each host.

Additionally, it maintains a mapping from host names to serial host identification numbers that is used for announcing host sets and all further communication concerning hosts.

When the arbitration daemon crashes, the first individual scheduler trying to send a command detects the crash. This scheduler then restarts the daemon, and all individual schedulers go through the `rsh` based authentication described above for `DMWMole` to establish a socket connection to the new daemon.

The individual schedulers and the daemon then cooperate in reconstructing the daemon state:

- Each scheduler sends its value of n_s to the daemon.
- Each scheduler sends its entire host database to the daemon and notes the new mapping (which will necessitate recomputing all $\mathcal{H}_j$).
- The daemon reestablishes connections to the `DMWMole` instances running on the hosts and queries their λ and μ values.
- Before requesting a host, individual schedulers send their current $\mathcal{H}_s$ set.

The only information that is not restored are the t_s and the number of jobs running on each host. The former information is only useful for tie breaking and thus not essential, while the latter tend to adjust to the correct values as jobs running at the time of the crash terminate and get reported to the arbitration daemon.

Queries using `DMWUmpire`

Like `DMWMole`, a connection to `DMWUmpire` is established by using `rsh` to run the program and get connection information back. The client can then interact with `DMWUmpire` using the one letter commands shown in table 5.2.

In addition to this functional interface, there is a *debugging interface*, shown in table 5.3, which allows users to observe scheduler activities on the whole cluster from the vantage point of the central coordinator.

H	Declare a host, either by name or by IP address. Returns the ID of the host.
I	Declare $\mathcal{H}_s$ as a text string of ones and zeroes.
W	Request a host $h \in \mathcal{H}_s$.
R	Declare that one of the jobs on a previously allocated host has terminated.
J	Declare preexisting n_s upon reestablishing contact after a crash of the arbitration daemon.
Q	Disconnect the scheduler from the `DMWUmpire`.
S	Shut down the `DMWUmpire`. This command is never generated by the scheduler, but can be sent manually.

Table 5.2: Scheduling commands accepted by the `DMWUmpire` daemon.

D	Write a list of all hosts known to the arbitration daemon, with their current load average and free memory.
C	Write a list of all clients connected to the arbitration daemon with their n_s.
L	Toggle logging function, where the daemon writes a detailed protocol of the load information and scheduling requests it gets, and of its resulting actions.

Table 5.3: Debugging commands accepted by the `DMWUmpire` daemon.

5.6 Job Migration

A *job migration mechanism* for `DMW` was designed and implemented for SunOS 4 and Solaris by Michael Buschauer and Marcel Steinmann in a term project [BS96] and refined and ported to various other UNIX variants by Felix Rauch [Rau96] in an internship at ISE AG.

Consistent with the requirements for the entire system stated in chapter 2, the migration mechanism was defined to cover a somewhat more limited scope than the mechanisms discussed in the literature:

- Jobs are migrated only between hosts running the same operating system version on binary compatible hardware architectures.
- Only open disk files are preserved, but no sockets, pipes, or other interprocess communication mechanisms.
- Both checkpointing and migration are initiated by external processes.
- Programs to be migrated are available in source form and thus can be linked with a migration library.

5.6.1 Basic Architecture of the Job Migration System

The process resident portion of the job migration system is contained in a library `ckpt_lib` which is linked with programs to be migrated.

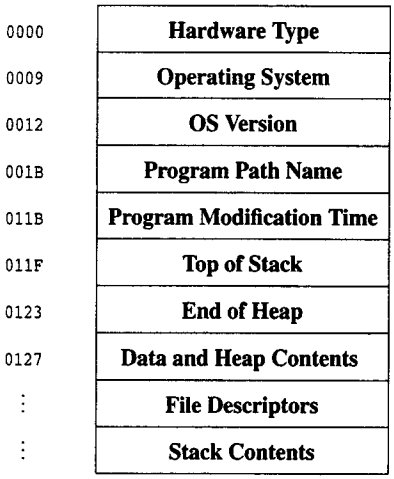


Figure 5.2: Organization of `ckpt_lib` checkpoint files

Initializing `ckpt_lib`

`ckpt_lib` has to be initialized with a call

```
#include "ckpt_lib.h"
    :
    :
ckpt_setup(argc, argv);
```

early in the execution of the program, passing the parameters to the program itself.

Tracking Open Files

`ckpt_lib` has to keep track of all open file descriptors and their read/write positions. For this purpose, `ckpt_lib` intercepts the `open()`, `dup()`, `dup2()`, and `close()` library routines to record the creation and destruction of file descriptors. These routines call through to the real system calls using `syscall(SYS_xxx, ...)`.

If a relative path name is passed to `open()`, `ckpt_lib` finds the absolute path to record by calling `getcwd()`.

Creating Checkpoints

Checkpointing is triggered by sending the process one of the signals SIGUSR1 or SIGUSR2. `ckpt_lib` executes a `setjmp()` call to preserve the CPU state and writes a checkpoint file as shown in Figure 5.2.

The checkpoint file does not contain the text segment, i.e., the program code itself, since this information is immutable and available from the original program binary when the job is resumed.

Restoring from a Checkpoint

When a program is restored from a checkpoint (either because of a host failure or after a migration) `ckpt_lib`

- performs some consistency checks on the checkpoint file,
- expands the stack by calling itself recursively (to guard against overwriting its own local variables during restoring),
- restores the saved process image,
- finally executes a `longjmp()` to the saved `jmp_buf`, resuming the program.

5.6.2 Platform Specific Considerations

While this basic model proved adequate for all target platforms, there were still significant differences between platforms, especially in the layout of process images. To identify these differences, the Condor [LLM88] source code often provided valuable guidelines.

Sun SPARC/Solaris

On many platforms, the buffered I/O call `fopen()` calls `open()`; on Solaris, it calls a functionally identical routine `_open()` instead. Therefore, `ckpt_lib` on Solaris has to intercept `_open()` as well as `open()`.

`dup2()` is not a system call on Solaris, so the original routine is not accessible with `syscall(SYS_dup2, ...)`. Instead, the dynamic C library is linked to the program and the `dup2()` routine from that library is called.

HP PA-RISC/HPUX

The process address space organization on HP-UX is rather different from other platforms, and presented some difficult challenges to porting. Consequently, the port of `ckpt_lib` to HP-UX cannot be considered to be more than a proof-of-principle implementation.

HP-UX is the only target platform where the stack grows toward higher addresses instead of lower addresses, but fortunately, this does not complicate implementation much.

However, the HP-UX address space architecture includes so-called "MMF" segments which need to be saved. We were unsuccessful at determining the number and location of MMF segments at runtime, so we had to resort to a highly inefficient approach [Rau96]: When creating a checkpoint file, `ckpt_lib` executes a `fork()` and immediately has the child process terminate itself, creating a core dump. The parent process then determines the location of MMF segments from the core dump and writes these areas in its own address space to the checkpoint file.

On HP-UX, it does not suffice for the buffered I/O routines to simply restore all data in the process image, so `ckpt_lib` keeps track of `fopen()` calls and during recovery,

open buffered streams are reopened with `freopen()`. Like Solaris, HP-UX has both an `open()` and a `_open()` routine.

IBM RS6000/AIX

The AIX implementation of `ckpt_lib` also proved to be problematic, as AIX is based on a shared library architecture with no convenient means of overriding system calls. To overcome this obstacle, a process using `ckpt_lib` forks itself and then inserts `trap` instructions into the shared library call stubs of the traced system calls. When the child process tries to call any of those library routines, the parent process is notified and instructs the child process with the `ptrace()` call to first call the corresponding routine in `ckpt_lib` before resuming the real system call.

Intel 80X86/Linux

The Linux implementation is simple compared to the preceding two platforms. The only problematic aspect of this implementation is that `ckpt_lib` is unable to create a checkpoint while the process is executing in a shared library, so checkpoint and migration signals may have to be sent repeatedly.

Like on HP-UX, the Linux implementation has to track `fopen()` calls and reopen the buffered streams on recovery.

5.7 Evaluation of the Job Migration Mechanism

While the work of Buschauer, Rauch, and Steinmann resulted in a functional migration mechanism, it was never integrated into the `DMW` system. Some of the reasons for this decision were of a pragmatical nature, concerning the limitations of the implementation regarding portability, maintainability, and performance. However, the main reason was that experience with practical projects led to increasing doubts about the suitability of a homogeneous, general purpose, migration approach for our system.

I shall discuss the practical issues in section 5.7.1, the fundamental concerns in section 5.7.2, and shall sketch a more promising approach to job migration in section 5.8.

5.7.1 Practical Issues with `ckpt_lib`

Performance

On most of the platforms supported, `ckpt_lib` is capable of generating checkpoint files in a reasonably straightforward manner. The overriding of library routines tends to slow them down somewhat, but since I/O calls are very rare in our simulators, this should not create any significant costs.

On two platforms, however, `ckpt_lib` has to create a new process for its operation. In the case of AIX, this should not have adverse performance effects: The `fork()` occurs at the beginning of the lifetime of the process when it is still small, and the parent process then takes little CPU and uses a small working set. For `HP/UX`, however, circumstances are very different:

- The `fork()` occurs late in the lifetime of the process, which is potentially hundreds of megabytes in size at that time.

- The entire process image has to be written to disk twice: Once for the core file from which the location of the MMF segments is determined, and once for the checkpoint file itself.
- The core file has to be read back for the examination. Due to its size, it is unlikely to have remained in the disk cache and has to be read back from disk or from an NFS server again.
- Migration is usually initiated when a host is short of physical memory and overloaded. Creating a new, huge process and generating large amounts of I/O under these circumstances will further exacerbate the problem.

Due to this awkward checkpointing process, writing checkpoint files on `HP/UX` would be prohibitively expensive for large processes. Clearly, `ckpt_lib` is not ready for practical use on this platform, and it is not clear how it could be made to work on `HP/UX`.

Another performance issue is the fact that, due to its intrusion into the C I/O system, `ckpt_lib` does not work with the native Fortran compiler on some platforms, forcing the use of `f2c`, which can be up to 50% slower than the native Fortran compiler [Rau96].

Maintainability

While `ckpt_lib` works on the specific platforms and operating system versions it was ported to, deploying it in a commercial system would present significant maintenance problems:

- The layout of process memory images is determined by largely undocumented parameters.
- The data and execution flow between the buffered and unbuffered I/O systems vary from platform to platform.
- In addition, modern operating system features like shared libraries and memory mapped files add further complexities to checkpointing.
- Solutions are not portable between operating systems, often are not even portable to newer releases of the same operating system, and their correctness is hard to verify.

Thus, integrating `ckpt_lib` into `DMW` would have created an unacceptable maintenance risk. In contrast, the only inherently platform dependent code in `DMW` is the resource measuring code in the `DMWMole` daemon, which

- Is concentrated in a single, small file.
- Is adapted from an external, free source (the `top` source distribution) which is highly likely to remain maintained by people familiar with the kernel internals as new operating systems and versions get released[3].

[3] cf. [Ray98]:

> Perhaps in the end the free-software culture will triumph not because cooperation is morally right..., but simply because the commercial world cannot win an evolutionary arms race with free-software communities that can put orders of magnitude more skilled time into a problem.

- Has a fallback strategy of resorting to the `rstat` code if for some reason the platform dependent code does not work.

In view of this, if any general purpose migration mechanism were to be integrated into DMW, it would be preferable to look into adapting an existing solution like Condor, which has been ported to many platforms and is actively maintained.

5.7.2 Fundamental Problems with General Purpose Migration

Checkpoint File Sizes

Checkpoint files written by a general purpose migration mechanism always have to save the entire process image, which for our applications leads to huge checkpoint file sizes. The LargeParallel application discussed in chapter 7, as an example, consists of 72 `dessis` (device simulation) jobs with process sizes between 80 and 140 megabytes. If one checkpoint for each of the jobs had to be stored simultaneously, the application, which currently takes about 250 megabytes of disk space to execute, would need more than 7.3 gigabytes of disk space.

Disk space requirements can be kept within more acceptable limits if checkpoint files are only written on demand for a job to be migrated, rather than at regular intervals. However, under such a strategy, checkpoint files are written at a time when the process is already exceeding available resources on the host or inconveniencing an interactive user. Writing the checkpoint file will generate large amounts of I/O. Furthermore, it requires accessing the entire memory image of the process, which in low memory situations is likely to generate large numbers of page faults. Thus, migrating a job will cause conditions on the host to *worsen* for several minutes.

Heterogeneity

A second fundamental drawback of general purpose migration mechanisms is that they are restricted to migrating jobs between hosts running identical operating systems (the systems described in section 3.4.3 are clearly not yet suitable for practical use).

This can lead to pathological situations on a workstation cluster consisting of some fast hosts with little memory running a hardware/OS combination A and some slower hosts with more memory running a hardware/OS combination B: Jobs get started predominantly on platform A, but after a few hours, they start thrashing and with homogeneous migration, it is impossible to transfer them to platform B where they could run to completion.

5.8 Application Specific Migration

The experimental results, and the doubts they cast on the suitability of general purpose, homogeneous migration, call for a new assessment of the case for *application specific* migration mechanisms.

5.8.1 Need for General Purpose Migration

`ckpt_lib` was designed as a general purpose mechanism mainly because we assumed that many different simulation tools would be able to profit from migration, and that the

effort required to equip all migration candidate programs with migration mechanisms would be prohibitive.

However, the experiments show that it is mainly the process and device simulators which create large, long running processes for which initial placement alone is inadequate while all other tools run only for a few minutes and are adequately managed with initial placement. Thus, application specific mechanisms would be needed for only a handful of programs.

5.8.2 Size of Files Representing the Computation State

Another insight from the experiments is that, contrary to our original assumptions, it is possible to express the computation state of simulation processes in a significantly more compact form than achievable by saving the entire memory image of the process: Both the `dios` and `dessis` simulators studied produce output and dump files only a few hundred kilobytes in size, even for processes requiring more than hundred megabytes of memory to run. Furthermore, these files are entirely independent of the platform on which they were produced.

The reason for this enormous difference in the size required to store essentially the same amount of information is that the in-memory representation of the data is optimized toward computation speed, while the format of the output file is designed for compactness. Furthermore, the iterative methods employed require storing the results of previous computation steps so that convergence of the computation can be determined [Feu98].

It therefore appears promising to design application specific checkpointing mechanisms based on the existing output or dump file format. Such a mechanism already exists to some extent in `dios`: To implement parametrization, a dump file can be written after a process step, which is then read in as a starting point for several further process simulations proceeding with different parameter settings.

5.8.3 Application Specific Checkpointing

To extend the existing support into a checkpointing mechanism, the programs have to be extended so the writing of dump files can be triggered *asynchronously* to the computation. This can be accomplished by installing a signal handler for some user defined signal which sets a global flag indicating that a checkpoint file should be generated. The computation can check this flag at convenient times (e.g., at the end of an iteration step) and write a checkpoint file if it is set. Developers of both `dios` [Str98] and `dessis` [vM98] have confirmed that such a mechanism would be possible to implement and that potential checkpoints, i.e., occasions at which the state of the simulator could conveniently be captured and restored, are closely spaced, within a few minutes of each other.

5.8.4 Migration Mechanisms

Depending on how short the desired preemption period should be, such a checkpoint mechanism can be employed in different modes:

Migration When a job should be migrated, a checkpointing signal is sent to it and the scheduler waits until the checkpoint file is written. Then, the job is terminated

and the scheduler looks for a new host to restart the job. At this point, it is able to improve on its original placement decision because it has additional information:

- The size of the job at the time it was terminated can serve as a new lower bound on the memory required after the restart.
- The load situation in the cluster will probably have changed since the start of the job, and thus a new placement might find a faster host.
- Furthermore, a new placement is able to take into account which machines are occupied by interactive processes.

Checkpointing A job can also be be sent checkpointing signals periodically, e.g., every 15 minutes. If migration is then considered necessary, the job can simply be terminated immediately and computation can be restarted from the last checkpoint file.

Given the compact size of checkpoint files discussed above, checkpointing is likely to require quite modest amounts of time and disk space, so the advantages of this strategy—greater robustness and the benefit of having jobs disappear on a machine the moment they are found to be inappropriate there—will probably outweigh the disadvantages—some wasted computation and additional I/O and disk space needs.

Leer - Vide - Empty

Chapter 6

DMW: Implementation

Pay no attention to the man behind the curtain!
Frank Baum, *The Wizard of Oz*

In this chapter, I shall discuss some implementation issues that are of importance to the overall software design of the scheduling system, but are not directly relevant to the question of distributed scheduling.

6.1 Basic Design Principles

DMW is implemented using the C++ programming language. While C++ has gained widespread popularity due to its wide availability, efficiency, and expressive power, it has become fashionable over the past few years in some circles to criticize C++ for numerous real or debatable flaws.

One important criticism is that C++ offers too many different ways to express the same concept, and too little guidance to navigate through the maze of possibilities. This charge is certainly justified, and it is thus important for successful C++ projects to decide on one coding style and to consistently adhere to it throughout the project.

In this section, I shall present the coding style used in DMW, starting with a discussion of programming-in-the-small issues like nomenclature of classes and variables and then proceeding to programming-in-the-large issues like the interaction of classes among themselves.

6.1.1 Nomenclature

In a large programming project with multiple programmers, there is a certain risk that two programmers independently pick the same name for two different global entities. While the C++ language standard will offer the *namespace* facility to control the visibility of global identifiers, this feature is not implemented yet on most of the C++ compilers we work with.

Therefore, DMW resorts to a traditional method of disambiguating identifiers: All globally visible names (i.e., global variables and class names) are prefixed with `DMW...`, which is a substring unlikely to be chosen in any other part of the program.

A similar potential for naming clashes exists for *enumeration literals*, i.e. the symbolic values for an enumeration type. Therefore, DMW avoids declaring enumerations with global scope and instead declares them inside of class definitions, which makes

Identifier Class	Spelling	Affix	Example
Class Names	Uppercase, BiCapitalization	DMW...	DMWJob
Member Functions	Uppercase, BiCapitalization		GetStartTime
Data Members	Lowercase, BiCapitalization		startTime
Global Variables	Uppercase, BiCapitalization	gDMW...	gDMWMolePath
Global Functions	Uppercase, BiCapitalization	DMW...	DMWRestoreJob
Static Variables	Uppercase, BiCapitalization	g...	gLocalPath
Static Functions	Uppercase, BiCapitalization		RshPath
enum Identifiers	Lowercase, BiCapitalization		ready
Preproc. Constants	All uppercase, underscores	DMW_...	DMW_HOSTDB_PATH
Preproc. Macros	Like functions		DMWLog
Header Guards	Like file name	..._H	DMWJob_H

Table 6.1: Naming conventions used in DMW source code

enumeration literals only accessible if their names are prefixed with the name of the class and thus disambiguated.

Furthermore, it helps readability to adopt a consistent set of conventions regarding capitalization of identifiers and whether multiple words in an identifier are combined with underscores (as in Host_Name) or *BiCapitalization* [Ray96] (as in HostName). Table 6.1 shows the conventions used in the DMW source code. Note that the preprocessor #define mechanism is used for macros in three different roles:

- Most parameterless macros are *constants*. Since C++ has a const mechanism, preprocessor constants are rarely used.
- Proper *preprocessor macros* usually have parameters. Like constants, they are rarely used, mainly to specify logging functions which are usually disabled.
- To prevent multiple inclusion of header files, each header files surrounds its declarations with a *header guard*:

```
#ifndef DMWJob_H
#define DMWJob_H
    :
    :
#endif // DMWJob_H
```

6.1.2 Existing Class Libraries

Many of the data structures used in DMW are quite stereotypical: Linked lists, hash tables, priority queues. While the C++ standard will define template libraries for such classes, their implementation for current generation C++ compilers is often incomplete or even nonexistent.

Therefore, DMW instead relies on the commercial *Rogue Wave Tools++* library to provide standard classes. While the version of the Tools++ library used by us is not very new and is not as elegantly designed as the standard C++ library, especially in the area of container iterators, it works quite well and doubtlessly has saved a considerable amount of work. All classes in the Rogue Wave library have names prefixed with RW....

6.1.3 Class Organization and Interaction Patterns

Successful object oriented projects often contain recurring patterns of classes and communicating objects. Gamma et al. in an influential book formalized the idea of *Design Patterns* [GHJV95] and compiled a catalog of the most useful patterns.

Design patterns were first proposed by Christopher Alexander et al. for the architectural design of cities and buildings. Alexander writes [AIS+77]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Applying this methodology to software construction, Gamma et al. write:

> Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. ... The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*
>
> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

DMW makes frequent use of these patterns, and I will use the names introduced by Gamma et al. in the subsequent discussion. Appendix C gives a short introduction to the design notation that I will use, which is also adopted from [GHJV95].

6.2 DMW classes

In this section, I shall present the most important classes in the DMW library in a manner resembling the *Literate Programming* style, pioneered by Knuth [Knu92]. Figure 6.1 shows the fundamental classes involved in scheduling and their relationships: A DMWSchedule contains a collection of interdependent DMWJobs and runs them. To make its placement decisions, the DMWSchedule has access to a DMWConstraintDB, a collection of DMWConstraints representing static scheduling constraints, and to a DMWHostDB containing a collection of DMWMonitors to keep track of the load conditions on the remote hosts.

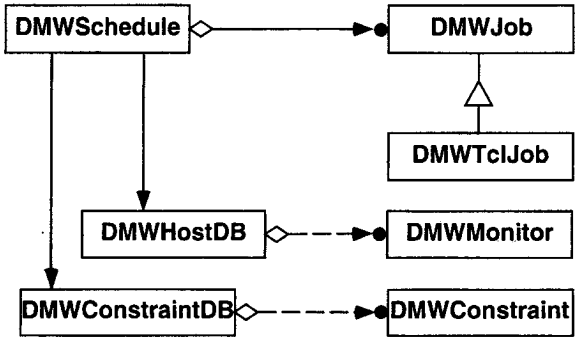


Figure 6.1: Classes Involved in Scheduling

6.2.1 DMWSchedule

A DMWSchedule is a collection of DMWJobs and their dependences. To simplify manipulation of DMWSchedules from Tcl code, DMWJobs are assigned a numerical ID within the schedule so that they can mostly be manipulated by that ID rather than directly by pointers.

AddJob adds a DMWJob to the schedule and returns the new job ID. RemoveJob removes a job or, optionally, an entire branch of the job dependence graph from the schedule. A number of functions translate between job IDs and the jobs themselves.

```
1   class DMWSchedule {
    public:
      typedef int DMWJobID;
      // Add a job to the schedule with the given local ID
5     // (or the next available ID if jobID is 0)
      DMWJobID AddJob(DMWJob * job, DMWJobID jobID=0);

      // Remove a job. If closure is TRUE, also remove all
      // jobs that depend on it
10    void RemoveJob(int jobID, RWBoolean closure = FALSE);
```

```
    // Associate between job IDs and the DMWJob objects
    DMWJob * operator[](DMWJobID jobID) const;
    DMWJobID FindID(DMWJob * job)  const;
    DMWJobID MaxJobID()            const;
```

Once the job IDs of the prerequisite and target jobs are known, dependences can be specified between them. Normally, *data dependences* are used to specify that the target job needs some of the results of the prerequisite job. A data dependence ensures that the target job is not started until the prerequisite job has terminated successfully.

Occasionally, users decide after the first few variants of a simulation have been completed that the simulation should be aborted. Therefore, it is sometimes useful to encourage depth-first execution order of the simulation tree. This is done with *priority dependences*, which ensure that the target job is not started until the prerequisite job has started.

```
    // Reasons for a dependence: data means target needs data
    // files written by prerequisite, priority just means the
    // user is more interested in fast results from prerequisite
    enum Dependence {dataDependence, priorityDependence};

    // Add a dependence
    void AddDependence(
            DMWJobID prerequisite, DMWJobID target, Dependence dep);

    // Remove dependence
    void RemoveDependence(DMWJobID prerequisite, DMWJobID target);

    // Are all dependences satisfied?
    RWBoolean       Eligible(DMWJobID jobID);
```

For traversing sets of jobs and dependences, DMWSchedule defines job and dependence *iterators*: Clients obtain instances of these classes and get the jobs or dependences themselves by repeatedly calling the `Next` member function of the iterator.

Gamma et al. [GHJV95] point out several advantages of using iterators:

- Iterators support variations in the traversal of the aggregate class: Various sets of jobs in a schedule can be traversed with the same interface, simply by substituting a different instantiation of DMWSchedule::JobIter.
- Iterators simplify the aggregate interface: Instead of five sets of traversal functions, DMWSchedule only needs to define two iterator classes and five iterator instantiation functions.
- Multiple traversals can be pending on an aggregate simultaneously.

```
    class JobIter {
    public:
      virtual RWBoolean Next();
      virtual DMWJobID  ID();
      virtual DMWJob *  Job();
      virtual ~JobIter();
    protected:
      JobIter()                         {}
    };
```

```
40    class DepIter {
      public:
        virtual RWBoolean  Next();
        virtual DMWJobID   Prerequisite();
        virtual DMWJobID   Target();
45      virtual Dependence Dep();
        virtual ~DepIter();
      protected:
        DepIter()                          {}
      };
50
      // Return all jobs
      JobIter * Jobs() const;

      // Return all jobs with fixed hosts
55    JobIter * HostedJobs() const;

      // Return jobs with a certain status
      JobIter * Jobs(DMWJob::Status status) const;

60    // Return prerequisites of a job
      DepIter * Prerequisites(DMWJobID jobID) const;

      // Return jobs dependent on a job
      DepIter * Dependents(DMWJobID jobID) const;
```

Simulations are usually started by calling `Run` on the schedule and periodically calling `Refresh` to update the status of running jobs and schedule further jobs. In some situations, it may make sense to schedule jobs one at a time by calling `Next` instead.

`Suspend` stops the execution of a simulation temporarily until `Run` is called again. `Terminate` kills all executing jobs and stops scheduling.

```
65    // Schedule a single job & return its ID
      virtual DMWJobID    Next();

      // Run schedule to completion or until we change our mind
      virtual void   Run();
70
      // Suspend scheduling until further notice. If stop is TRUE,
      // stop all running jobs, else let them run to completion
      virtual void   Suspend(RWBoolean stop);

75    // Do periodic maintenance. Return FALSE if either all
      // processes are done or scheduling is suspended.
      virtual RWBoolean Refresh(int maxJobs, DMWJobID * lastJob);

      // Terminate all jobs by any means necessary
80    virtual void Terminate();
    };
```

6.2.2 DMWJob, DMWTclJob

Each program to be executed in a simulation is represented by an instance of class DMWJob. Often, it is necessary to perform minor coordination tasks between jobs, and starting a new job (possibly even scheduling it on a remote host!) for such tasks would be highly uneconomical. Since GENESISe uses Tcl/Tk [Ous94] extensively, a very simple and convenient solution was to introduce a class DMWTclJob to allow such minor jobs to be specified in Tcl. DMWJobs and DMWTclJobs share the same abstract interface and behave identically regarding dependence analysis, but the latter are always executed synchronously in the current process as soon as they are eligible to run.

Figure 6.2 shows state transitions for instances of DMWJob: Jobs get created in `ready` state, enter the `running` state upon being started with `Run()` and finally enter `done` state. The other states signal problems:

- If the scheduler cannot properly start a job, the job is marked as `fubar`.
- If the job terminates with an error exit code, it is marked as `failed`.
- If the job is terminated by the scheduler (usually upon user intervention), it is marked as `terminated`.
- Finally, if a job is temporarily stopped with a signal, it is marked as `stopped`.

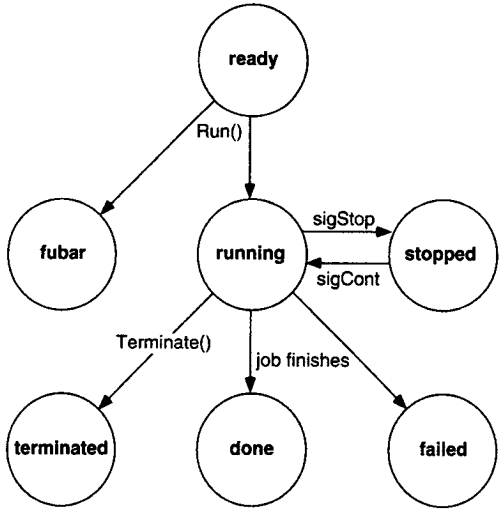


Figure 6.2: DMWJob State Transitions

```
class DMWJob {
public:
  // Current status of a job
  enum Status     {
    ready,        // Object created, but job not started yet
    fubar,        // Attempt to start job failed
    running,      // Job alive, last we heard
    stopped,      // Job alive but stopped
    terminated,   // Job terminated with Terminate() call
    done,         // Job ended voluntarily with successful status
    failed};      // Job ended by itself with unsuccessful status

  // Returns the status of a job, and, if it is done,
  // also the exit status
  virtual Status JobStatus(int * exitstatus);

  // Start a job. The job must have been in ready state, and after
  // this call, will be in either fubar or running state.
  virtual void Run();

  // Terminate the job by any means necessary
  virtual void Terminate();
```

While the job is running, signals can be sent to it and its priority can be manipulated.

```
  // The signals that may be sent to a job.
  // These four are sufficient.
  enum Signal     {sigKill, sigInt, sigStop, sigCont};

  // Send a signal
  virtual void SendSignal(Signal signal);

  // Set the nice value of the job
  virtual void SetNice(int nice);
```

The standard output and error streams are available as file descriptors, operating by default in nonblocking mode.

```
  // Returns the file descriptors for reading
  // the output and error streams.
  int    Output() const;
  int    Error()  const;

  // Specify blocking or non-blocking I/O for the output streams
  // Default is nonblocking. Return old state.
  virtual RWBoolean  SetBlocking(RWBoolean blocking);
```

The host on which a DMWJob runs is usually not specified at creation time, but assigned by the scheduler just before the job is run.

```
  // Change the host a job should run on
  void SetHost(const char * host);
  const RWCString & Host();
```

While DMWTclJobs support this entire interface, their behavior is much simpler: They support only the ready, done, and failed states and Run() directly executes the job. Neither the file descriptor calls nor the process control calls do anything useful. To distinguish between job types, the Trivial() member function returns TRUE for Tcl jobs and FALSE for normal jobs.

```
    // Delaying this job would be pointless
    virtual RWBoolean Trivial();
}; // class DMWJob
```

6.2.3 DMWMonitor, DMWHostDB

To make placement decisions, the scheduler needs to communicate with all of the potential target hosts to get up to date performance data. All of the details of communicating with the actual performance measuring agents are encapsulated in the class DMWMonitor, which presents a clean front end for answering questions about a single host. The DMWHostDB ensures that only one DMWMonitor needs to be created for each host.

The fundamental operation on a DMWMonitor is Query, which asks about some data value related to the host. Some of these values (the hostname and IP address of the host) are available locally, while others may require communication with the host, possibly interposing some caching mechanism to avoid requesting the same value repeatedly in short intervals. For uniformity and portability reasons, both requests and replies are translated into character strings.

```
class DMWMonitor {
public:
  static const char * NoOp;
  static const char * LoadAverage;
  static const char * MemoryStatistics;
  static const char * HostName;
  static const char * HostAddr;

  // Determine response to one of the above queries.
  // TRUE if successful.
  virtual RWBoolean Query(const char * query);

  // If previous query was successful, return pointer to response.
  operator const char*();

  // Combine Query() and result, return "" if unsuccessful
  const char * operator[](const char * query);
```

When many queries have to be made, it is preferable to separate the sending of the query from reading the response. The Prepare() method sends the query, and if subsequently Query() is called, only the response is read.

```
  // Give monitor advance notice of Query
  virtual RWBoolean Prepare(const char * query);
```

For a further performance improvement, the reading of the response can be changed to only read as much data as is available immediately and concatenate the portions until the entire line is read. The Advance() member function reads as much data as

is available and returns 1 if the response is complete, −1 if the request failed, and 0 if another call to `Advance()` should be made in the future.

```
20      // Slowly read.
        // Return 1 for success, -1 for failure, 0 for retry
        virtual int Advance(const char * query);
```

If communication with a host fails, it should be removed from consideration for scheduling. The `Sabotage()` member function disables the host, initially for 5 minutes. If at that time, the host is not available yet, the timeout is doubled, until availability is checked once every hour. The `Disfunctional()` member function indicates whether the host is currently considered out of order.

```
        // Host does not seem to work, mark it as unavailable
        // for a longer time
25      RWBoolean Disfunctional();
        void Sabotage();
```

While a freshly scheduled job starts up on a host, load measures tend to be inaccurate, underestimating the sustained load that the job will put on the host. Therefore, hosts that receive a job are *embargoed* for some time and only will be reconsidered for scheduling after load measures can be considered reliable again.

```
        // Mark the host as off limits for this scheduling cycle
        void SetEmbargoed(RWBoolean embargo);
        RWBoolean Embargoed();
```

When using the centralized, `DMWUmpire` based scheduler, all the local schedulers must agree on a common identification code for each host, assigned by the umpire.

```
30      // Manipulate Umpire ID
        int     ID();
        void    SetID(int newID);
    };
```

The *host database* in DMWHostDB is the collection of all hosts on which jobs could possibly be allowed to run, each represented by an instance of DMWMonitor. Since it never makes sense to have more than one instance of DMWHostDB in a program, this class is organized in a *singleton* pattern: By calling the static `HostDB` method, clients get the only instance, creating it if necessary.

The host database can operate in two different modes: In *eager* mode, the set of permissible hosts and their parameters is read from a *host database file* as soon as the instance of DMWHostDB is created. In *lazy* mode, instances of DMWMonitor are created on demand when a previously unknown host name is specified.

After a change of host parameters, the instance of DMWHostDB is destroyed with `Reset` to force the program to build a new host database.

```
    class DMWHostDB {
35  public:
      static DMWHostDB * HostDB(RWBoolean lazy);
      static void        Reset(RWBoolean reload, RWBoolean lazy;
```

Instances of DMWMonitor are requested by specifying either the name or the IP address of the host. If the umpire is used, DMWHostDB is also able to return the host designated in the host database file for running the umpire.

```
    DMWMonitor * GetMonitor(const RWCString & host);
    DMWMonitor * GetMonitor(DMWNetHost & host);
    DMWMonitor * GetUmpire()
```

If the umpire is used, DMWHostDB defines a mapping between DMWMonitors and umpire IDs. Since this mapping may change with little notice when the umpire is restarted, IDStamp returns a *time stamp* incremented at every mapping change so that program components relying on the mapping can quickly check whether they are up to date.

```
    DMWMonitor * operator[](int i);
    void         SetID(DMWMonitor * host, int id=-1);
    void         ResetIDs();
    RWBoolean    HasIDs();
    static long  IDStamp();
```

Hosts returns an iterator object for iterating through all currently defined hosts.

```
    class HostIter {
    public:
      virtual RWBoolean     Next();
      virtual DMWMonitor *  Host();
      virtual ~HostIter();
    protected:
      HostIter()                        {}
    };

    HostIter * Hosts();
  };
```

6.2.4 DMWRelay

The DMWRelay class serves as a local *proxy class* for an instance of a remote DMWMole or DMWUmpire daemon, encapsulating the details of communication. DMWRelay is usually not used directly, but as a building block for a subclass.

On construction, the DMWRelay instance receives the host and the executable path of the daemon to which it is to connect.

```
  class DMWRelay {
  public:
    DMWRelay(DMWNetHost * host, const char * path);
    virtual ~DMWRelay();
```

Connect() starts the connection, performing the entire handshake if block is TRUE, otherwise returning as quickly as possible. Connected() returns whether the connection succeeded yet.

```
    // Start connection and perform handshake.
    // FALSE means failed, TRUE means check Connected() and call again
    // if necessary.
    virtual RWBoolean    Connect(RWBoolean block=TRUE);

    // Check whether we're connected
    RWBoolean            Connected()       { return status>=s_conn; }
```

`Interact()` performs a synchronous query, sending a text message and waiting for a response in the form of one line of text.

```
// Try to send a message and get a response
virtual RWBoolean    Interact(char * msg);
```

Making a remote query involves a round-trip through a TCP connection and thus may be quite time consuming, especially considering that the same query usually has to be sent to dozens of hosts. Querying n hosts with an individual round-trip time t^{RT} with the synchronous `Query` interface will take

$$T_{sync} \approx n * t^{RT}_{avg}.$$

As the number of hosts grows, this strategy is increasingly likely to take too long, causing upper level layers of the scheduler to consider the `DMWUmpire` daemon running the queries to have crashed. If any of the hosts in the host database are for some reason not working, this likelihood turns into a virtual certainty.

Sending all queries out in parallel and gathering replies as they arrive would be much more efficient, requiring in the ideal case only

$$T_{async} \approx t^{RT}_{max}$$

independent of n, if t^{RT} is much longer than the processing time. However, due concerns about the complexity of implementing such a solution, I initially rejected this strategy.

As an intermediate strategy, attempting to preserve the simplicity of the synchronous strategy while trying to reap the performance benefits of the asynchronous solution, the query and response phases can be separated: In a *preflying* phase, all queries are sent out in parallel, and responses are then collected one by one.

The `Prefetch()` member function sends a query without waiting for the response.

```
15    // Send message but don't wait for answer. Follow with Interact
      virtual RWBoolean    Prefetch(char * msg);
```

The preflying strategy still results in a time

$$T_{prefly} \approx t^{RT}_{max}.$$

and works as well as the asynchronous approach if hosts are all reliable. If some of the hosts fail to reply or take an unreasonably long time to reply, however, the preflying approach will cease to give satisfactory results, while an asynchronous approach degrades much more gently, and, in fact, can exploit the degradation of response times to give scheduling preference to more lightly loaded hosts, as described by Theimer and Lantz [TL88].

Traditional techniques for asynchronous multiplexing in POSIX systems include:

- Multitasking, by forking one process for each query connection.
- Multithreading, by creating one lightweight thread for each query connection.
- Multiplexing the query sockets with the `select()` call.

Each of these techniques has important drawbacks: Multitasking would require creating dozens of new processes, which is highly undesirable for a daemon. Multithreading has a lot of promise for the future, but is not yet sufficiently portable across platforms. `select()` is economical and portable, but its use for multiplexing would have required destroying the encapsulation of the DMWMonitor class. Furthermore, tests showed that under some circumstances, `select()` may report a socket as readable but a subsequent `read()` on the socket may block anyway.

Therefore, multiplexing is performed by employing one response buffer per query and filling it with *nonblocking read operations*, which always return immediately, whether data is present or not. After the query is sent with `Prepare()` as in the prefly-ing approach, the `Advance()` member function performs a nonblocking read, returning 1 if the read operation delivered the rest of the response line, 0 if more data remains to be read, and -1 if the connection failed or timed out.

```
    // Look whether more data arrived. Return
    //   1 if the reply has arrived
    //   0 to keep trying
20  //  -1 if we gave up
    virtual int          Advance();
```

All DMW daemons identify themselves with a version number in the form of a decimal date like 19971002 so clients know which protocol requests are supported by the daemon. This version number is available in the `version` field.

```
    long         version;
};
```

6.2.5 DMWConstraint, DMWConstraintDB

Some of the software sold to a customer may for technical, administrative, or licensing reasons only run on a subset of all available hosts. The *constraint database* maintains for each program the set of hosts on which it is allowed to run.

Hosts are added and removed from this set by calling `Allow` and `Deny`.

```
1   class DMWConstraint {
      void                Allow(DMWMonitor * host);
      void                Deny(DMWMonitor * host);
```

The set of permissible hosts can either be traversed with an iterator or obtained as a bitset of umpire IDs. In the latter case, clients must be careful to recalculate the bitsets if the umpire is restarted, as IDs are not guaranteed to remain constant between restarts.

```
5     const RWBitVec &      EligibleSet();

      DMWHostDB::HostIter * Hosts();
    };
```

The constraint database in DMWConstraintDB is the collection of all DMWConstraints. Like the host database, the constraint database obeys a singleton pattern: There is only one instance of DMWConstraintDB, which is obtained by calling the `ConstraintDB` member function and rebuilt by calling `Reset`.

```
class DMWConstraintDB {
public:
  static DMWConstraintDB * ConstraintDB();
  static void              Reset(RWBoolean reload);
```

`GetConstraint()` returns the set of permissible hosts for some program. If no DMWConstraint for this program currently exists and `strict` is FALSE, a new instance with a default set (usually permitting access to all hosts) is created.

```
  DMWConstraint * GetConstraint(const RWCString & program,
                                RWBoolean strict = FALSE);
```

`Constraints` creates an iterator to traverse the entire constraint database.

```
  class ConstraintIter {
  public:
    virtual RWBoolean     Next();
    virtual DMWConstraint *  Constraint();
    virtual ~ConstraintIter();
  protected:
    ConstraintIter()                        {}
  };

  ConstraintIter * Constraints();
};
```

6.3 Scheduler Factory Classes

As discussed previously, GENESISe is designed to support both the fully distributed and the arbitrated scheduler simultaneously, which is achieved by making both of the schedulers subclasses of DMWSchedule:

DMWRStatSchedule, named after its use of the `rstat` protocol, implements the fully distributed, load average based, scheduler.

DMWUmolSchedule, named after its use of the `DMWUmpire` and `DMWMole` daemons, implements the arbitrated, load average and memory availability based, scheduler.

As is evident from these descriptions, each of these schedulers also cooperates with a different implementation of the DMWMonitor class:

DMWRStatMonitor uses the `rstat` protocol to get the load average of a remote host.

DMWUmolMonitor launches a `DMWMole` daemon on a remote host to get information about its load average and available memory.

The creation of matching schedule and monitor classes is coordinated through the *abstract factory* class DMWFactory, as shown in Figure 6.3.

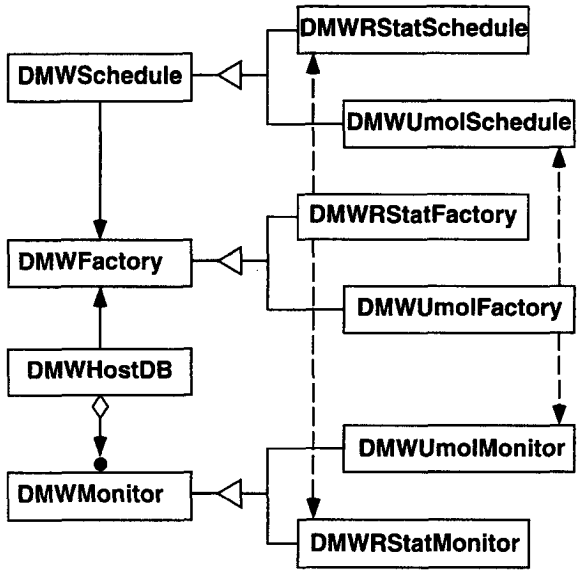


Figure 6.3: The DMW Class Factories

Instead of directly creating a DMWRStatSchedule or a DMWUmolSchedule, clients call the `NewSchedule` method of a DMWFactory object (which has been instantiated as either a DMWRStatFactory or a DMWUMolFactory).

```
1   class DMWFactory {
    public:
      virtual DMWSchedule    *NewSchedule();
```

The schedule object keeps a pointer to the factory, and passes it to the host database when requesting a monitor object. To create monitor objects, the host database calls the NewMonitor method of the factory.

```
    protected:
5     friend class DMWSchedule;
      friend class DMWHostDB;

      virtual DMWMonitor     *NewMonitor(
                DMWNetHost * host,
10              const RWCString  & arch,
                const RWCString  & os,
                double      weight,
                RWBoolean   disabled);
    };
```

Thus, a DMWUmolSchedule is guaranteed to always receive DMWUmolMonitors, and a DMWRStatSchedule will always receive DMWRStatMonitors, provided that the host database is reset every time the scheduling method is changed.

6.4 Daemons

While most of the scheduler is executing on behalf of the user on the host that the scheduler front end runs on, some of the functionality is delegated to daemons executing on remote hosts:

- The `DMWLaunchPad` daemon, described in section 5.2, controls jobs. Each job scheduled by `DMW` is managed by an instance of `DMWLaunchPad`.
- The `DMWMole` daemon, described in section 5.4, provides statistics of load averages and memory utilization on a host. If the arbitrated scheduler is used, an instance of `DMWMole` is started on each host in the host database.
- The `DMWUmpire` daemon, described in chapter 4, arbitrates hosts among schedulers, deciding which scheduler gets to start a job on which host. If the arbitrated scheduler is used, one instance of `DMWUmpire` is started on the arbitration host designated in the host database.

The implementation of daemon processes raises some interesting issues. Daemons must be:

- *Economic* in their memory and CPU consumption.
- *Robust* in handling multiple clients, dealing with the operating system, and rejecting manipulation attempts from malicious users.
- As *portable* as possible.

In the following sections, I shall discuss some of the issues relevant to the `DMW` daemons.

6.4.1 Conjuring a Daemon

UNIX takes steps to terminate processes still executing after the user who started them logs out, and makes it possible for users to easily send signals to processes they started. However, this is not desirable for most daemons, which need to keep executing independent of the user who started them. Therefore, a daemon process must gain immunity from a number of user and system generated signals, all of which are related to UNIX *terminals*[1]:

- Each process has a *controlling terminal*, which it inherits from its parent process. When a user logs out, all processes with his terminal as their controlling terminal get sent a *hangup signal* (`SIGHUP`).
- Processes are organized into *process groups*. One process group per terminal is considered the *foreground process group*. When a user types the interrupt, quit, or stop characters (often set to be `Control-C`, `Control-\`, and `Control-Z`, respectively), the *interrupt* (`SIGINT`), *quit* (`SIGQUIT`), or *stop* (`SIGTSTP`) signal gets sent to all processes in the foreground process group.

[1] This term is used here for a software abstraction which may represent physical hardware terminals or software concepts such as `xterm` windows or `telnet` sessions.

- When a process reads from a terminal to whose foreground process group it does not belong, it is sent a *tty input signal* (`SIGTTIN`), which by default causes the process to stop.
- Under some circumstances, when a process writes to a terminal to whose foreground process group it does not belong, it is sent a *tty output signal* (`SIGTTOUT`), which also causes it to stop by default.

To avoid being sent any of these signals by accident, a daemon process has to make sure on one hand not to associate itself with any controlling terminal and on the other hand not to perform I/O to any terminal. The first of these objectives is served by creating a new process with `fork()`, having the parent process exit, and calling `setsid()` in the new process. `setsid()` makes the process the sole member of a new process group and dissociates it from any controlling terminal. The second objective is served by closing all file descriptors which could be associated with terminals and not opening any new terminal devices.

After these steps are taken, a daemon should be secure from all accidental signals, although its owner and the superuser can still send it any signals by issuing explicit `kill` commands.

6.4.2 Client Authentication

In the `DMWMole` and `DMWUmpire` daemons, daemon startup is combined with client authentication. The daemon code proper and the authentication stub are combined into a single program which, upon being started

- Checks whether this daemon already runs on this machine.
- If so, prints connection information for the daemon.
- If not, starts the daemon and prints the connection information.

Additionally, the security concept in section 5.1 requires that the connection information be obtained with a method not accessible to processes not running on the local host. This objective is served by connecting to the daemon with a *local domain socket*[2], an interprocess communication method between processes running on the same host where sockets get bound to file names instead of IP addresses.

On startup, therefore, the daemon process

- Opens a local domain socket.
- Attempts to bind the socket to the file name `"/tmp/DMWLock/<daemon>"`, where `<daemon>` stands for the name of the daemon.
- If this attempt fails, a daemon is already running, so the process *connects* to that file name (bound by the socket of the daemon instance) instead, prints the data it receives, and exits.
- If the attempt to bind succeeds, the process `forks`. The child process then becomes the daemon, while the parent process reconnects to the local socket, prints the the data it receives, and exits.

[2] *Also frequently referred to as a UNIX domain socket*

While the daemon is executing, it *listens* for connections on two sockets: The local socket with a fixed address, and an internet socket with a randomly chosen address and password. If a client (which will, in fact, always be another instance of the daemon process) connects to the local socket, the daemon

- Accepts the connection.
- Prints a line containing
  - The daemon's version (date).
  - The TCP port of the internet socket.
  - The password for the internet socket.
- Closes the connection.

No authentication on these connections is necessary (or possible) because they can only originate from processes on the same machine, which must already have passed `rsh` or `ssh` authentication to run.

If a client (i.e., an individual scheduler or a `DMWUmpire` daemon) connects to the internet socket, the daemon waits for the password to be given and, if it is correct, accepts commands on that connection.

6.4.3 Handling Multiple Clients Robustly

For reliability and security reasons, the `DMWMole` and `DMWUmpire` daemons must be able to

- Accept as many concurrent clients as possible.
- Handle client commands fairly, without blocking indefinitely for any client.
- Handle (deliberately[3] or accidentally) malformed input safely.

For this purpose, the daemons declare an array of *client objects*, defining a *state machine* for each file descriptor on which a client could connect:

```
1   struct Client {
      enum {
        Disconnected = 0,
        Connecting   = 1,
5       Connected    = 2,
        DamageControl= -1
      } active;
      char buffer[14];

10    RWBoolean HandleRead();

      void HandleCommand(char * cmd);
    };

15  Client  gClients[FD_SETSIZE];
```

[3] Some of the most severe security breakdowns, such as the 1988 Internet Worm [SRE+89] incident, were caused by deliberately sending excessively long input lines to poorly written internet daemons. The Internet Worm exploited this bug in the `fingerd` daemon, but despite the enormous publicity the incident received, a similar bug was discovered almost seven years later [CER95] in the widely used NCSA http daemon!

Each file descriptor starts in the `Disconnected` state. When the daemon `accepts` a connection for that descriptor, the state is upgraded to `Connecting` and then, when authentication is accomplished, to `Connected`.

The daemon keeps a *file descriptor set* of all connected descriptors and periodically checks them for incoming data with a `select()` call. For each descriptor which has data, the daemon calls the `HandleRead()` member function, which appends data (and a 0 byte) to the buffer and then checks it for a newline character.

If no newline character is found and the buffer is full (with one 0 byte at the end which is never overwritten), the client diagnoses that an overly long line has been written (both `DMWMole` and `DMWUmpire` have a natural limit for the length of correct commands) and puts the state machine into `DamageControl` mode, in which all input up to the next newline character is discarded.

If a newline character is found, it is replaced with a 0 character, the complete line is passed to `HandleCommand()` and the line is then removed from the buffer. `HandleRead()` returns `TRUE` to indicate to the daemon that it should check for further commands even before any further data arrives.

This arrangement is robust and safe:

- It is capable of accepting as many client connections (`FD_SETSIZE`) as the OS is theoretically able to accommodate at all, which reduces the risk of a *denial of service attack* by starting many connections.
- By declaring the array statically, it avoids dynamically allocating and deallocating client data structures, which in long running daemons may lead to *heap fragmentation* and continuous growth of the daemon process size.
- It never blocks waiting for incomplete command lines and thus is robust against crashing or malicious clients and network induced delays.
- It is robust against arbitrarily malformed input.
- Through the use of the `select()` call, daemons waste no CPU time with *busy waiting* but are blocked by the operating system as long as no client activity occurs.

Part III

Experiences with DMW

Leer - Vide - Empty

Chapter 7

Measurements

> The process of earning a doctorate does not acculturate you to solving other people's problems, as most employers would want. Instead, it encourages you to keep elaborating on your thesis research. At least it leads you to believe that all the world's a research laboratory, equipped for your personal benefit.
>
> P. J. Plauger, *The Physicist as Programmer*

In this chapter, I present experimental results from running a selection of process and device simulation problems, as well as some synthetic benchmarks to highlight specific issues. I shall start by briefly introducing the simulation tools used and the benchmark problems chosen.

To understand the performance of a global scheduler, it is important to first understand how the *local scheduler* on a host executes jobs started on that host. I shall therefore present various experiments analyzing the behavior of jobs scheduled by the local UNIX scheduler (Solaris 2, unless stated otherwise) on a single host: Section 7.3 examines the behavior of a single large job on a host, section 7.4 examines the coexistence of multiple large jobs on single processor and multiprocessor hosts, and section 7.5 examines the interaction between background jobs and CPU intensive jobs started by interactive users.

Section 7.6 then proceeds to analyze the results of scheduling the benchmark problems on a workstation cluster. Finally, to demonstrate the differences between the distributed and the arbitrated implementation of DMW, I shall present a scenario where multiple "users" submit simulations concurrently in section 7.7.

7.1 Simulation Tools Used

Dios

`dios` is a general purpose process simulator for simulating the application of fabrication processes to 1D, 2D, and certain classes of 3D structures. `dios` supports a wide variety of materials and structure sizes. If requested, it can interactively display the simulation in a graphical window.

Dessis

`dessis` is a 1D/2D/3D device and system simulator, simulating the electrical and thermal behavior of semiconductor devices and circuits by solving:

- The external circuit and contact equations
- The Poisson equation
- The continuity equations
- The heat transport equation
- The hydrodynamic carrier transport equations

7.2 Benchmark Problems

Table 7.1 summarizes the benchmarks measured:

- The `LargeSequential` benchmark, consisting of a single job characterized by a very long running time and high memory demands, is the `dios_pblocos` example project distributed with `dios`.
- The `SmallParallel` benchmark, consisting of a few rather small jobs, is the `nmos,process` example program for `GENESISe`.
- The `LargeParallel` benchmark, characterized by a large number of rather large simulation jobs (36 dios jobs, 72 dessis jobs), is due to Alexander Höfler [Höf97].

Figure 7.1 illustrates the structure of the process and device simulation jobs in the three problems.

Benchmark	dios Jobs				dessis Jobs			
	#	t [h]		μ [k]	#	t [h]		μ [k]
LargeSequential	1	11.00	3.50	115000	—	—	—	—
SmallParallel	3	0.04	0.02	28016	3	0.02	0.01	52256
LargeParallel	36	2.50	0.92	75972	72	2.50	0.60	96640

Table 7.1: Summary of benchmark problems. 1D and very short 2D dios jobs used for parametrization are omitted. Running times are median CPU times for a Sun SPARCSTATION S10-61 (60 MHz SuperSparc processor) (left column) and a Sun Ultra Enterprise 3000 (4 x 250MHz UltraSparc processors) (right column).

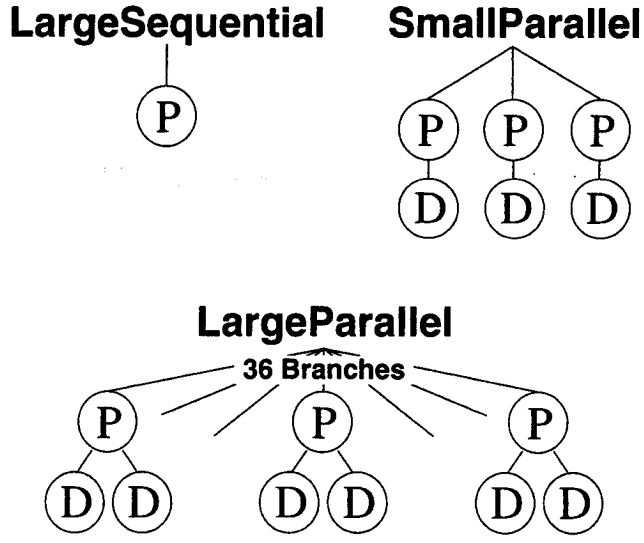


Figure 7.1: Dependences of process (P) and device (D) simulation jobs in the three benchmark problems.

7.3 Local Scheduling: Single Job

In this section, I shall examine the behavior of a single large job run on a host, by first presenting an experiment where abundant amounts of memory are available and then contrasting it with an experiment with more constrained memory.

Finally, I shall discuss correlations in the resource consumption of related job in a parametrized simulation.

7.3.1 Abundant Memory

When a large simulation job is executed on a machine with few other processes and sufficient memory, it will behave as shown in Figure 7.2, which shows the execution of the LargeSequential benchmark on a 60MHz Sun SPARCSTATION 10 with 192 megabytes of physical memory.

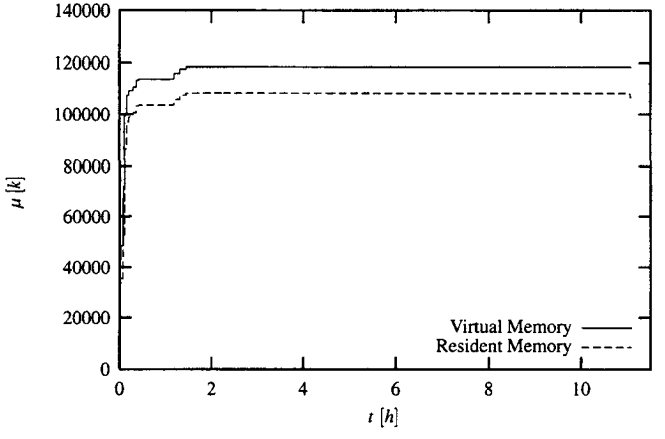


Figure 7.2: Memory consumption of a job executed on a host with sufficient memory (60MHz Sun SPARCSTATION 10 / 192M)

It can be observed that

- The job takes slightly more than 11 hours of computation.
- The *virtual memory size*, the region of memory allocated to the job, rapidly grows to about 118500 kilobytes. It never shrinks.
- The *resident memory size*, the portion of the virtual memory space that is actually in physical memory, grows in parallel with virtual memory, but never exceeds 106500 kilobytes.

Figure 7.3 shows the startup phase in some more detail. It is evident that the job essentially reaches its full virtual memory size after about 9.5 minutes.

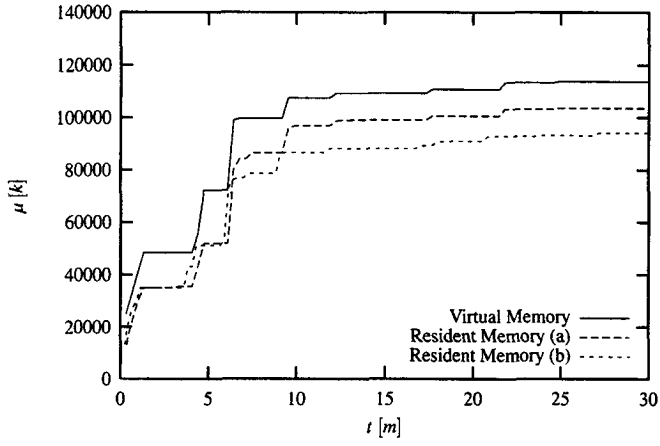


Figure 7.3: Startup phase of jobs in Figure 7.2 (a) and Figure 7.4 (b). Times for (b) are adjusted to compensate for the slower clock rate.

7.3.2 Constrained Memory

Figure 7.4 shows the LargeSequential benchmark again, this time executing on a dual processor 40MHz Sun SPARCSTATION 10 with 128 megabytes of physical memory. Given the resident size established in the previous experiment, the job should fit into physical memory, but just barely so. The figure shows that

- The resident memory size always is smaller than in the experiment with abundant memory.
- On several occasions, resident memory size drops sharply and then recovers somewhat.

In the following two sections, I shall discuss these two phenomena in some more detail.

Smaller Resident Memory Size

As Figure 7.3 shows, the job starts rather similar to the job in Figure 7.2, up to a virtual memory size of 100000 kilobytes. After that, the resident set size seems to grow more slowly than in the experiment with abundant memory.

This phenomenon is caused by a combination of two different mechanisms: Pages are added to the resident memory set by *demand paging* [GC94, Section 3.7], which allocates a physical memory page for a virtual memory page when it is referenced. They are removed from the resident memory set by *page stealing* [GC94, Section 3.8], a periodical background scan removing pages from memory (saving their contents to disk if necessary) if they were not referenced since the last scan.

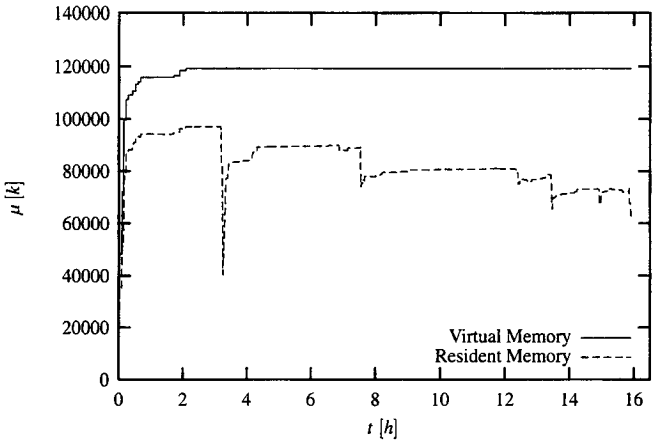


Figure 7.4: Memory consumption of a job executed under occasional thrashing conditions (40MHz Sun SPARCSTATION 10 / 128M)

Demand paging is driven by the memory reference pattern of the process, which is very similar for the two experiments. Page stealing, however, is driven by the memory situation on the machine: The activity of the page stealing daemon increases as memory gets more scarce, and may stop entirely if there are large amounts of free memory.

Therefore, the *growth* of resident memory is similar in the two experiments, but once the resident memory size approaches 80000 kilobytes, the page stealing daemon in the second experiment increases its activity to *shrink* resident memory simultaneously. The combination of these two effects results in the slower resident memory growth shown in Figure 7.3.

Thrashing Episodes

Figure 7.5 shows the first incident of resident memory drop in more detail, adding the CPU share of the job (note that since execution is on a dual processor workstation, that share cannot exceed 50%) and the load average of the host.

The plot shows that the resident memory size starts to drop after the job has run for three hours and 11 minutes. The job was started at 20:24 in the evening, so the incident starts at 23:35. It turns out that this is the time at which a daily backup script is started on the host by the `cron` OS facility. The backup is rather memory and CPU intensive, but on this day takes only a few minutes.

Due to the increased pressure on the OS memory allocation system, the page stealing daemon intensifies its activities and reduces the resident memory set by more than 55000 kilobytes. It turns out, however, that many of those pages were in fact part of the working set of the process and are needed again soon. Consequently, CPU share of the process drops temporarily to less than 5% as the process has to request its pages back. After 12380 seconds, some 15 minutes after the beginning of the incident, the

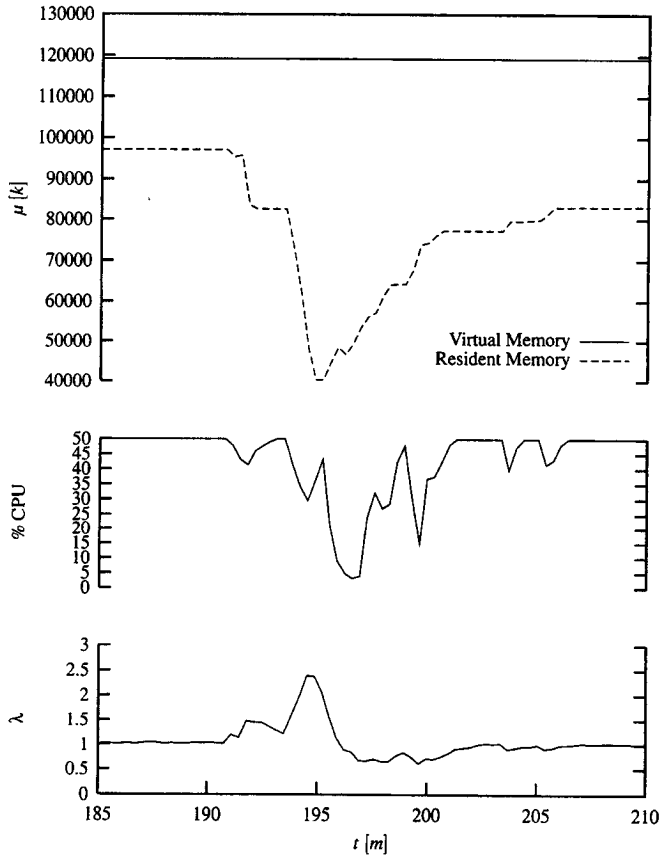


Figure 7.5: Details of the first thrashing episode in Figure 7.4

process finally is able to continue business as usual. It turns out that approximately 14000 kilobytes of the removed memory were indeed not used, and the process is able to proceed with a significantly smaller resident set size than before.

Note that the CPU use of the backup process has hardly any influence on the CPU share of the job here: The load average exceeded the number of CPUs only during one minute of the incident and never exceeded 2.4, so if there had been sufficient memory, the job would still have had more than 80% CPU share.

Overall, the local UNIX scheduler on the host shows that it is quite capable of dealing with moderate long term memory pressure by the adaptive activation of the page stealing daemon to recognize pages not used over long periods of time, with virtually no adverse effects on the job: Despite the memory pressure, the job took approximately as long to complete as the job run with abundant memory (taking into account the difference in clock frequencies).

However, once excessive memory pressure forces the page stealing daemon to rescan memory too quickly, it increasingly considers pages as unreferenced which are in fact part of the working set of a process and soon have to be brought back into memory. The resulting slowdown of the process, and of its memory reference frequency, further exacerbates this problem, and performance becomes unacceptable. In this experiment, the second memory intensive task only ran for a few minutes, so the slowdown was not too serious (and the incident occurred late at night, so interactive users were not affected), but if the second job is another lengthy simulation, the thrashing can persist for several hours, as section 7.4.2 will show.

7.3.3 Variations Among Related Simulation Jobs

Another area of interest is the behavior of related jobs in the same simulation—whether it is possible to predict the resource consumption of a new simulation job from the observed resource consumption of previously executed jobs. For this purpose, I now take a closer look at the jobs in the LargeParallel benchmark.

Table 7.2 summarizes the running times t and virtual memory requirements μ of the 36 `dios` jobs in the project. The jobs represent all combinations of

- Three different values for the implantation interstitial factor.
- Three different values for the implantation lateral standard deviation.
- Four different values for the gate length.

Those parameters are varied in the order given above, i.e., jobs 14063, 14071, and 14079 vary the interstitial factor while keeping the other two parameters constant, job 14191 varies the lateral standard deviation while keeping the other two parameters to the same value as in job 14063, and so on.

As Figure 7.6 shows, there is a fairly good correlation ($r > .95$) between the running time and the virtual memory size of the jobs. While this relation does not help to predict either of these unknown quantities, it supports the findings in section 4.5.2 that the memory demands of a job cannot be determined by observing it initially for any finite amount of time.

Table 7.3 summarizes the running times t, virtual memory requirements μ, and input grid sizes μ_g of the 72 `dessis` jobs in the project. A device simulation is performed for the devices resulting from each of the 36 `dios` simulations, with two different values for the body voltage parameter.

Node	t [m]	μ [k]	Node	t [m]	μ [k]	Node	t [m]	μ [k]
14063	43:30	68544	16239	51:27	69600	18415	67:34	80992
14071	45:07	69112	16247	55:16	70592	18423	77:48	81720
14079	46:36	68928	16255	53:31	70736	18431	77:33	86880
14191	51:08	68552	16367	55:58	75952	20207	89:22	116312
14199	51:56	68560	16375	57:01	75992	20215	93:25	116040
14207	51:21	68528	16383	56:53	76008	20223	94:52	115664
14319	53:00	73616	18159	55:18	80568	20335	109:28	116272
14327	56:16	73536	18167	59:59	81488	20343	103:40	116104
14335	56:45	73536	18175	61:09	81880	20351	104:48	115840
16111	46:57	69456	18287	64:38	81448	20463	106:38	139184
16119	46:02	70328	18295	66:15	80960	20471	111:47	116480
16127	49:17	70232	18303	65:50	81160	20479	109:33	115680

Table 7.2: `dios` jobs in LargeParallel benchmark (250MHz Sun Ultra Enterprise 3000).

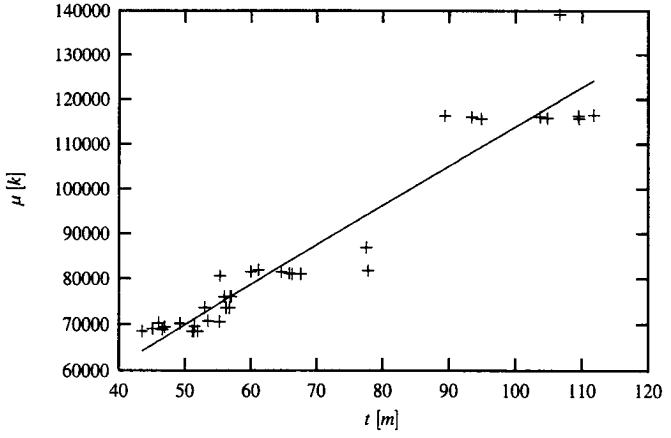


Figure 7.6: Memory use vs. running time for the LargeParallel `dios` jobs in table 7.2

Node	t [m]	μ [k]	μ_g [k]	Node	t [m]	μ [k]	μ_g [k]
168766	24:07	88384	453	217918	43:19	103376	575
168767	23:57	88384	453	217919	43:48	103376	575
168862	22:49	87984	452	218014	43:20	103376	575
168863	23:21	87984	452	218015	42:58	103376	575
168958	27:54	89344	452	218110	43:46	103376	575
168959	28:44	89344	452	218111	42:43	103376	575
170302	24:07	88384	453	219454	43:06	103376	575
170303	24:36	88384	453	219455	43:36	103376	575
170398	22:47	87984	452	219550	43:31	103376	575
170399	22:31	87984	452	219551	44:01	103376	575
170494	28:07	89344	452	219646	43:57	103376	575
170495	28:55	89344	452	219647	43:48	103376	575
171838	22:45	88384	453	220990	43:44	103376	575
171839	22:41	88384	453	220991	43:14	103376	575
171934	21:26	87984	452	221086	44:14	103376	575
171935	21:52	87984	452	221087	43:05	103376	575
172030	26:51	89344	452	221182	44:06	103376	575
172031	27:05	89344	452	221183	44:11	103376	575
193342	30:58	89904	454	242494	93:42	146712	958
193343	32:02	89904	454	242495	95:37	146712	958
193438	31:57	89896	453	242590	95:34	146968	959
193439	31:27	89896	453	242591	102:18	146968	959
193534	26:44	88728	453	242686	90:49	146752	959
193535	27:06	88728	453	242687	97:52	146752	959
194878	31:05	89904	454	244030	94:16	146712	958
194879	31:29	89904	454	244031	94:57	146712	958
194974	31:52	89896	453	244126	96:51	146968	959
194975	31:54	89896	453	244127	117:02	146968	959
195070	26:48	88728	453	244222	91:18	146752	959
195071	27:18	88728	453	244223	97:32	146752	959
196414	30:13	89904	454	245566	92:06	146712	958
196415	30:58	89904	454	245567	94:04	146712	958
196510	30:52	89896	453	245662	95:17	146968	959
196511	31:17	89896	453	245663	99:09	146968	959
196606	26:09	88728	453	245758	90:55	146752	959
196607	26:44	88728	453	245759	101:55	146752	959

Table 7.3: `dessis` jobs in LargeParallel benchmark (250MHz Sun Ultra Enterprise 3000).

Figure 7.7 shows that again, there is a fairly good correlation ($r > .99$) between the running time and the virtual memory use of the jobs.

However, a closer look at the data shows that in fact, there is an even better correlation ($r > .999$) between the *input grid size* and the virtual memory use of the jobs, as shown in Figure 7.8. This relation, furthermore, is much more useful, as the input grid size is known *before* the job is run and thus, in this example, could be used to predict the memory demands of a job before it is placed on a host.

Of course, this result cannot be generalized to *all* device simulations: The virtual memory demands of dessis simulations depend considerably on the simulation parameters, and is only that closely correlated to input grid size in this example because all of the device simulations were essentially run with identical parameters and the varying of the voltage parameter had no influence on memory demand.

Nevertheless, projects like the LargeParallel example, where essentially identical device simulations are run to determine the effects of variations in process simulation parameters, appear to be common enough that exploiting such a relationship in a scheduler enhanced with application specific knowledge might be worthwhile.

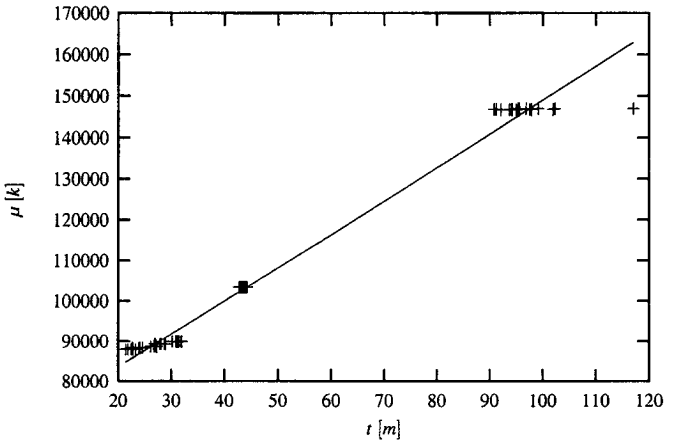


Figure 7.7: Memory use vs. running time for the LargeParallel `dessis` jobs in table 7.3

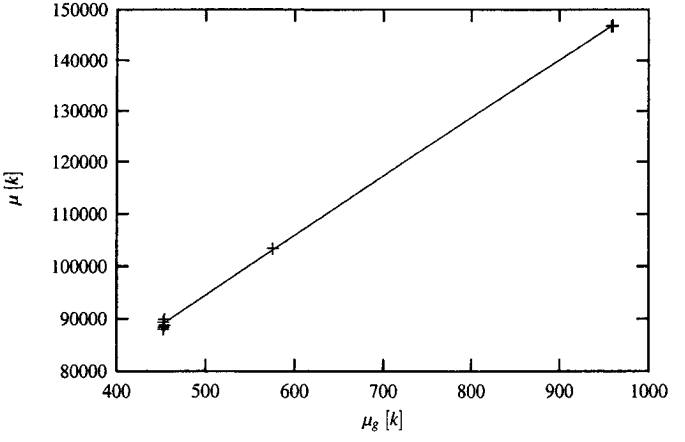


Figure 7.8: Memory use vs. grid size for the LargeParallel `dessis` jobs in table 7.3

7.4 Local Scheduling: Multiple Jobs

In this section, I shall discuss interactions between multiple jobs on a host. As in section 7.3, I shall first consider a situation where no memory shortages occur and then turn to the case of memory contention between multiple long running jobs.

7.4.1 Benefits of Multiprogramming

Since simulation jobs spend some portion of their time doing I/O, it might be tempting to assign more jobs to a host than its number of CPUs to assure that while one job is doing I/O, another job can use the CPU. This use of *multiprogramming* to improve CPU utilization has been employed in local scheduling since the 1960s [Tan87, section 1.2.3], so the original scheduling policy for `DMW` allowed scheduling more jobs on a host than the number of CPUs.

However, when I presented this policy at a conference [NCR97], Livny [LBRT97] challenged it, thinking it doubtful that the jobs would do a significant amount of I/O.

To study the effects of multiprogramming on the overall execution time, I ran the first 16 `dios` jobs from the LargeParallel benchmark on a 4 processor workstation, varying the degree of multiprogramming (i.e., the number of jobs being run simultaneously) from 1 to 16.

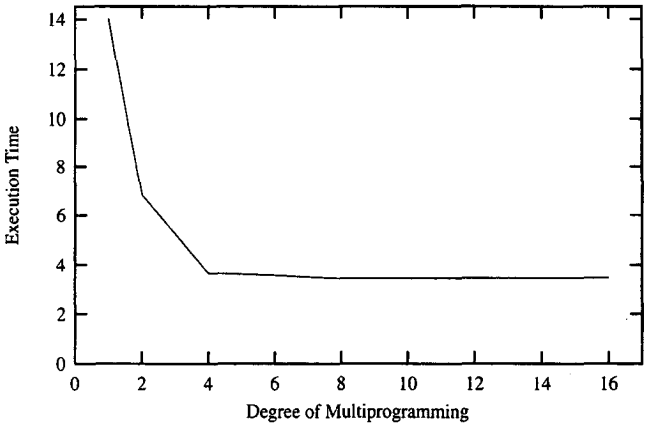


Figure 7.9: Running time vs. degree of multiprogramming (4 x 250MHz Sun Ultra Enterprise 3000)

The results of this experiment are shown in Figure 7.9, and they show that Livny's objection was indeed correct. It is evident that there are enormous performance benefits in scheduling as many jobs as the number of CPUs, but there are virtually no further benefits from scheduling more than 4 jobs.

Thus, scheduling more jobs than the number of CPUs is, at best, a performance

neutral practice. In fact, however, this policy is almost certainly *harmful* for two reasons:

- Scheduling too many jobs commits these jobs to one host for a long period of time, preventing the excess job from taking advantage of another host becoming available during that time.
- Every additional job on a machine increases memory usage on the machine, and once memory is overcommitted, performance deteriorates massively, as I will show below in more detail.

7.4.2 Long Term Thrashing

In section 7.3.2, thrashing episodes only lasted for a few minutes because, except for the simulation job, only short running programs were run during the simulation. If, however, there are several long running programs present, thrashing may persist for hours or even days.

Figure 7.10 shows the memory consumption of two `LargeSequential` jobs started simultaneously on the same host used for the experiment in section 7.3.2 (2 × 40MHz Sun SPARCSTATION 10, 128M of RAM).

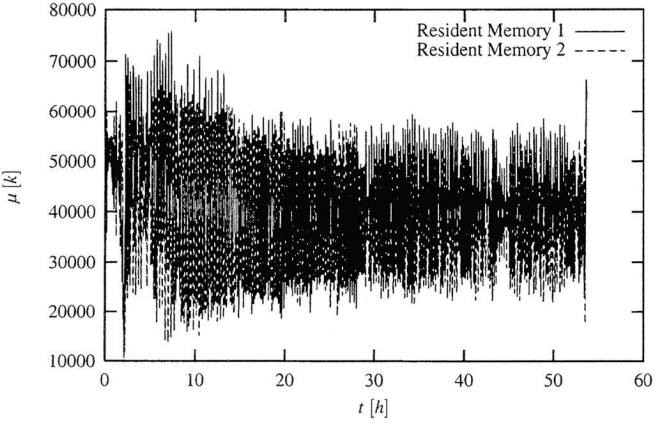


Figure 7.10: Massive thrashing as a consequence of running two memory intensive jobs

It is obvious that massive, continuous thrashing occurs. When examined in detail in Figure 7.11, it turns out that the seemingly chaotic system in fact exhibits a highly regular pattern of behavior, with both processes increasing and decreasing their memory allocation in 20 minute cycles.

Clearly, such behavior is undesirable: Instead of completing in 16 hours, as they would have if sufficient memory had been available, or in 32 hours, as they would

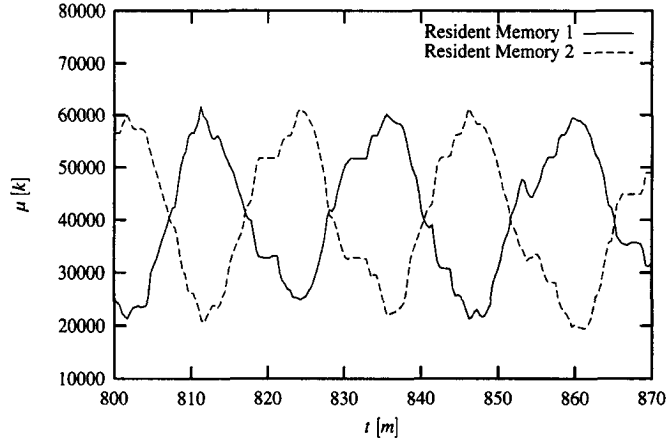


Figure 7.11: A detailed view of a passage from Figure 7.10

have if they had been run one after the other, the two jobs take more than 53 hours to complete and render the host entirely unusable for interactive users during that time.

While in this experiment, the situation was deliberately created by manually starting the two large jobs, it is far from unrealistic: Absent information about the memory needs of jobs, the global scheduler *should* usually consider scheduling two jobs on a two processor machine, and obviously, without a job migration mechanism, there is no choice but to either let processors go idle or risk thrashing like in this example.

7.5 Local Scheduling: Influence of Nice Parameter

To prioritize jobs, all UNIX variants offer the `nice` mechanism, which adds a bias to the priority calculated by the operating system. Typically, process priorities are recalculated a few times each second based on the consumed CPU time t for the process as

$$p_{new} = \delta p_{old} + (1 - \delta)(p_{user} + w_t t + \text{nice}),$$

where the *decay factor* δ in some UNIX variants is fixed, in others is dependent on the load average. Processes with the *lowest* numerical priorities then get the opportunity to run, which increases their accumulated CPU time t until they exceed the priority value of another process, which is then preferred to them.

As long as only a single CPU bound process runs on a host, its `nice` value is irrelevant, but as soon as several processes are eligible to run, those with lower `nice` values get a higher percentage of the CPU than those with higher `nice` values. This allows the scheduler to reduce the interference of the jobs it starts with foreground jobs by assigning high `nice` values to its jobs.

Figure 7.12 shows the influence of the `nice` value on various UNIX systems. Two identical, CPU bound processes were run, with the "foreground process" being run with a `nice` value of 0, while the `nice` value of the "background process" was varied between 0 and 19.

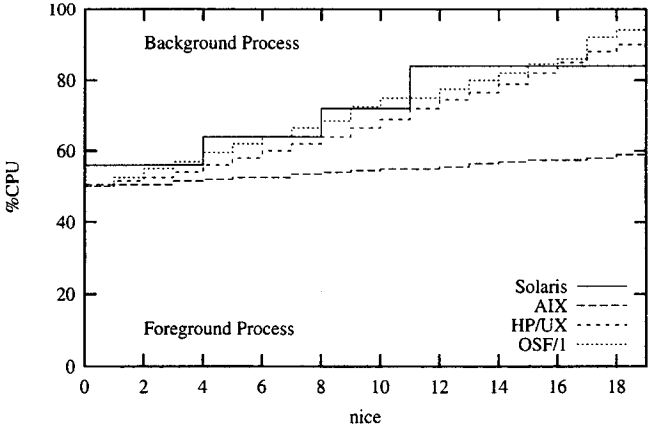


Figure 7.12: Influence of `nice` parameter on CPU share

The HP/UX and OSF/1 measurements correspond closely to what one would expect according to the formula above: As the `nice` value is increased, the CPU share of the background process drops gradually to about 10%, while the share of the foreground process increases.

While the Solaris behavior is similar to HP/UX and OSF/1, CPU share does not

increase continuously with each increase in the `nice` value, but appears to increase in bigger increments, with a similar overall effect.

The AIX local scheduler, though, differs significantly from the other schedulers. Even at the highest settings of the `nice` parameter, the foreground process does not get more than 60% of the CPU, rendering the use of `nice` largely ineffective for the protection of interactive processes.

7.6 Global Scheduling: Single User

After all this discussion of local scheduling, it is time to see the global scheduler in action. For this series of experiments, I again executed the LargeParallel benchmark, but this time on a collection of a few dozen Sun workstations instead of just using a single workstation.

Table 7.4 lists the workstations used, along with their single processor performance relative to a 60MHz SuperSparc processor (determined by running a tight loop of multiply-accumulate operations on each workstation).

Figure 7.13 shows a first attempt, running the LargeParallel example on the full host database, with all hosts limited to 1 job, except for `zinal` with 4 jobs and `blanche` with 2 jobs[1].

A first glance at the results suggests that DMW has been successful to some extent, reducing a single processor running time of almost 100 hours (for a 250MHz UltraSparc-II processor) to less than 11 hours on the 34 processors used and thus turning a simulation formerly taking more than half a week into literally an overnight job.

A closer examination, however, reveals some less favorable scheduler behavior:

- The *rate of parallelism*, shown at the bottom of Figure 7.13, remains high for the first 3 hours of the experiment only and then drops considerably, although not even all jobs have been started and the experiment proceeds for another 8 hours.
- The `dios` simulations (marked by light triangles) hold up the `dessis` simulations (marked by dark triangles) dependent on them, but DMW does not schedule some of the `dios` simulations until 2.5 hours into the experiment (e.g. on `stoos` and `copenhagen`).
- Some of the hosts chosen are so slow (e.g. `blanche`) that simulations started on them hold up the experiment although no resource contention occurs on the machine during their execution.
- As a consequence of the previous two phenomena, some of the `dessis` jobs are only started after the experiment has already run for almost 8 hours.

It is also evident that not all of the hosts listed are assigned jobs, but this is caused by the hosts being busy with other tasks and thus represents desirable behavior.

These observations suggest two obvious improvements to the execution of the experiment:

- The default, depth first ordering, ordering of jobs in the schedule (choosing a `dessis` job over a `dios` job when both are eligible to run), is often useful, making complete results of the first simulation branches available to the user as quickly as possible. If, however, the user wants to execute an entire schedule as quickly as possible, a breadth first ordering ordering is more efficient, making jobs eligible to run as quickly as possible.
- The inclusion of some of the slower hosts may actually *increase* turnaround time for a single simulation, although the increased throughput it provides is normally useful. To obtain the fastest turnaround for a single experiment, it may therefore be useful to omit some of the slower hosts.

[1] Given the limited amount of memory on `blanche`, it would not be advisable to run more than 2 jobs.

Host	Memory	Processor		β
maipo	1024M	2 × 200MHz	UltraSparc	3.2
zinal	512M	4 × 250MHz	UltraSparc-II	2.7
arkadin	640M	250MHz	UltraSparc-II	2.7
bamako	640M	250MHz	UltraSparc-II	2.7
stoos	256M	250MHz	UltraSparc-II	2.7
katita	576M	2 × 167MHz	UltraSparc	1.8
satchmo	576M	2 × 167MHz	UltraSparc	1.8
smorodina	576M	2 × 167MHz	UltraSparc	1.8
celine	320M	2 × 167MHz	UltraSparc	1.8
copenhagen	320M	2 × 167MHz	UltraSparc	1.8
etna	320M	2 × 167MHz	UltraSparc	1.8
galadriel	320M	2 × 167MHz	UltraSparc	1.8
gwaihir	320M	2 × 167MHz	UltraSparc	1.8
tabor	320M	2 × 167MHz	UltraSparc	1.8
hohgant	512M	167MHz	UltraSparc	1.8
fluela	256M	167MHz	UltraSparc	1.8
mtblanc	256M	167MHz	UltraSparc	1.8
mythen	256M	167MHz	UltraSparc	1.8
tonto	256M	167MHz	UltraSparc	1.8
buin	192M	143MHz	UltraSparc	1.5
zauberberg	192M	143MHz	UltraSparc	1.5
horex	192M	75MHz	SuperSparc-II	1.3
rapanui	160M	75MHz	SuperSparc-II	1.3
badile1	160M	110MHz	uSparcII	1.1
badile2	160M	110MHz	uSparcII	1.1
badile12	160M	110MHz	uSparcII	1.1
cassini	128M	110MHz	uSparcII	1.1
laudinella	128M	110MHz	uSparcII	1.1
tsa	128M	110MHz	uSparcII	1.1
nair	96M	110MHz	uSparcII	1.1
acme	224M	75MHz	SuperSparc-II	1.1
stradivari	192M	60MHz	SuperSparc	1.0
alvier	160M	60MHz	SuperSparc	1.0
ela	160M	60MHz	SuperSparc	1.0
fujiyama	160M	60MHz	SuperSparc	1.0
blanche	192M	6 × 50MHz	SuperSparc	0.9

Table 7.4: Host database used for global scheduling experiments.

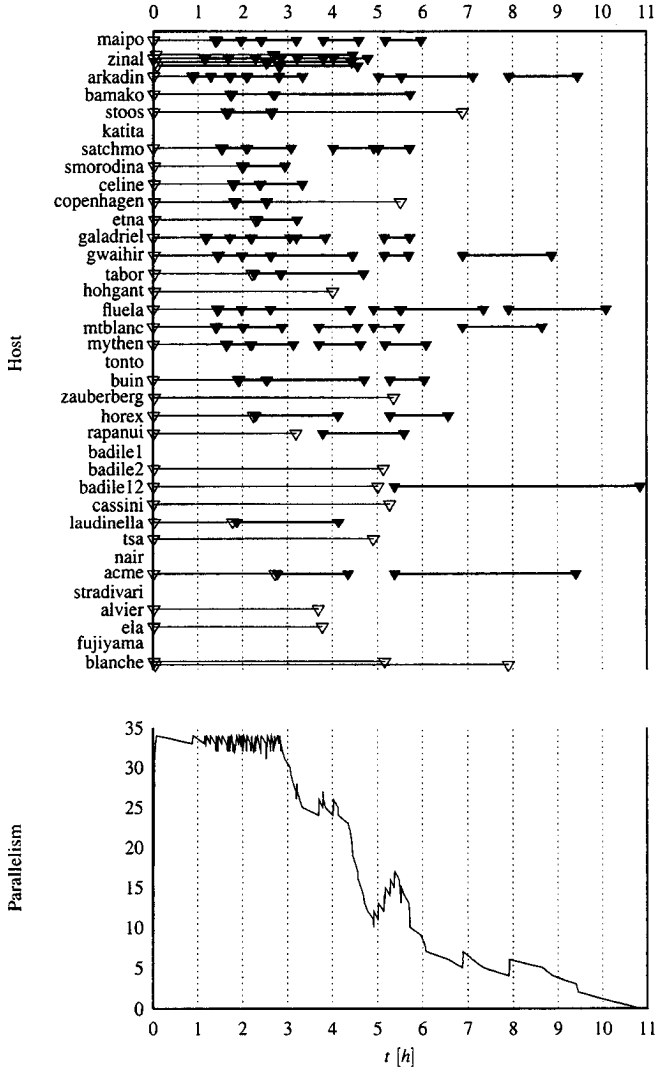


Figure 7.13: Global scheduling experiment, full host database, depth first precedence

Figure 7.14 shows the result of implementing these ideas by reordering the schedule in breadth first order and omitting all hosts with $\beta < 1.3$, but configuring all remaining multiprocessor hosts with their job limit set to the number of processors.

Although the number of usable processors has dropped to 25, this execution is significantly faster again, completing in just over 8 hours. This improvement is due to various factors:

- `smorodina` and `satchmo`, two of the faster hosts, are allowed to use both processors (the other multiprocessor hosts, except for `zinal`, are partially busy and therefore do not use their full job limit).
- Thanks to the breadth first reordering, `dios` jobs are scheduled first.
- Omitting the slower hosts, combined with the reordering, ensures that all `dios` jobs complete within the first 5 hours of execution.
- This ensures a steady supply of ready to run jobs: As opposed to the previous run, this time no processors have to go idle as long as any jobs remain in the schedule at all (`celine` and `tabor` were busy with other tasks before participating in the execution), and parallelism remains near the maximum until the last job in the schedule is started.

Nevertheless, it is again the slowest host which ultimately holds up completion: If the last job on `rapanui` had been run on another host instead, or if it could have been migrated during its execution, the simulation could have completed another hour earlier.

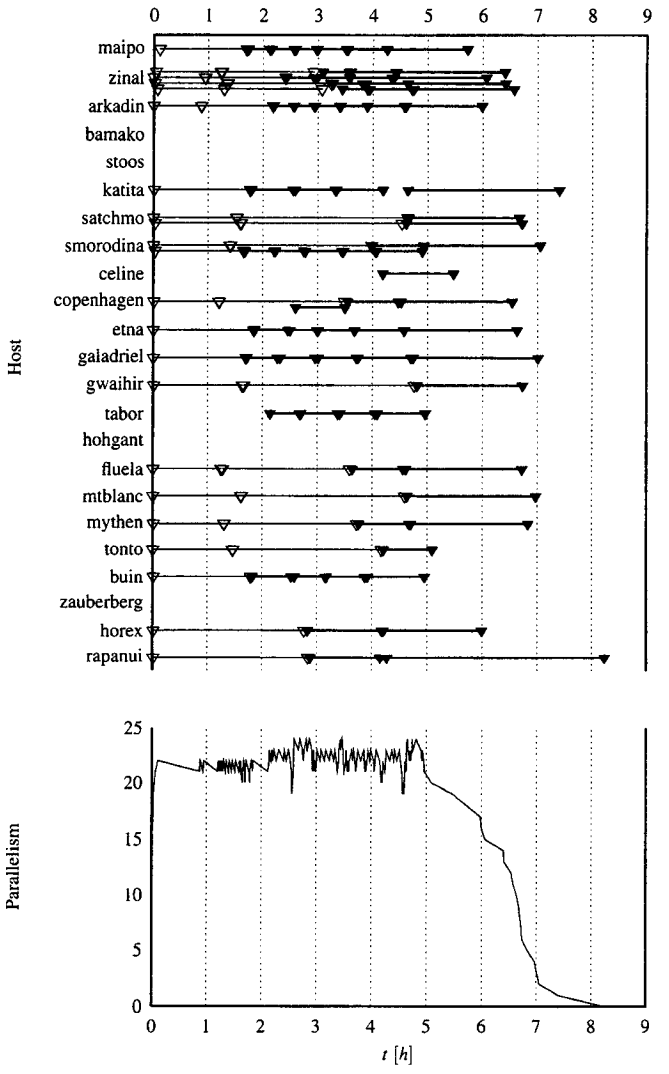


Figure 7.14: Global scheduling experiment, abridged host database, breadth first precedence

7.7 Global Scheduling: Multiple Users

So far in this chapter, all experiments were performed under the assumption that only a single user is using the scheduler simultaneously. What happens, however, in the more realistic case of multiple simultaneous users?

Figures 7.15 and 7.16 show such a situation, simulated by simultaneously executing three copies of the SmallParallel benchmark, enabling hosts `gwaihir`, `buin`, `horex`, and `rapanui` with a limit of 1 job each.

Figure 7.15 shows the results when run under the old, fully distributed version of the scheduler, while Figure 7.16 shows the results under the new, centrally arbitrated version of the scheduler.

It is evident from the figures that the `UMol` scheduler performs significantly better in this experiment:

- The `RStat` scheduler is not capable of enforcing global job limits.
- Even worse, the `RStat` scheduler has the tendency, discussed in section 4.3.2, that every individual scheduler considers the same host the least loaded at any given time, and thus all individual schedulers tend to simultaneously start jobs on the same host.
- In contrast, the `UMol` scheduler enforces global job limits and allocation fairness among individual schedulers.

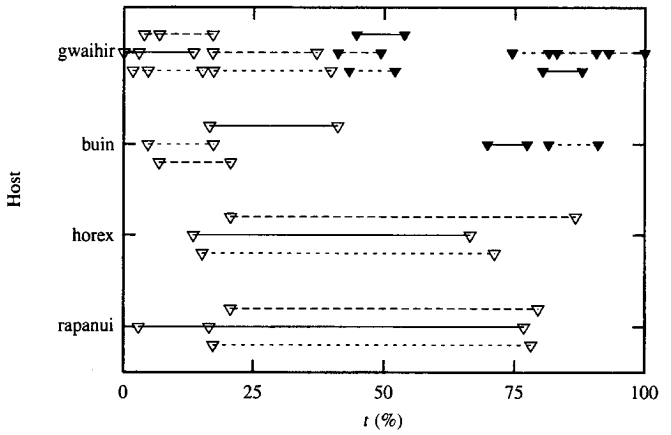


Figure 7.15: Multiple users running simulations under the fully distributed (`RStat`) scheduler

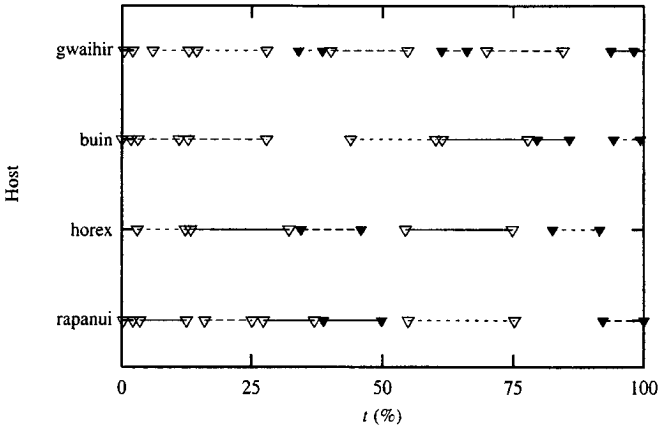


Figure 7.16: Multiple users running simulations under the centrally arbitrated (`UMol`) scheduler

Chapter 8

Conclusions

In this utterance of falsehood upon falsehood, whose contradictories
are also false, it seems as if That which I uttered not were true.
Aleister Crowley, *The Book of Lies*

At this point, I would like to summarize the experience gained with the operation of DMW and with the measurements in the preceding chapter, juxtaposing those results with the objectives defined in chapter 2. I shall conclude with a discussion of open problems and possible future improvements to the scheduler.

8.1 Results

In discussing the results, I shall start with a general set of quality criteria for a global scheduler and an assessment how well DMW and schedulers described in the literature meet them. In a second section, I shall evaluate DMW in the light of the objectives established at the beginning of the work. Finally, I shall discuss the issue of fully distributed vs. partially or fully centralized schedulers.

8.1.1 Quality Criteria for Global Schedulers

Typical modern workstation based environments have considerable spare processing resources that can be exploited to speed up time consuming computations. Due to the rapidly fluctuating availability of the spare resources, the task of finding idle workstations for a computation is best delegated to a global scheduler. Ideally, such a scheduler should strive to be:

Non-intrusive: i.e., it should install on a workstation with no changes to the hardware or the operating system, no installation of privileged software and no changes to system configuration files.

Non-obtrusive: i.e., it should never interfere with the computations of users using a workstation interactively.

Platform independent: i.e., it should run on a wide range of computing platforms.

Application independent: i.e., it should run with every application that can run on any of the supported platforms.

All currently existing distributed schedulers have shortcomings in some of these areas:

- Many of the research systems described in the literature are highly intrusive, being based on exotic operating systems or requiring considerable modifications to the host operating systems. Due to the lack of standard cross platform resource tracking facilities on POSIX platforms, most global schedulers are forced to rely on their own mechanisms for resource tracking, necessitating intrusive procedures on many platforms.
- All schedulers relying solely on initial placement of jobs are sometimes obtrusive.
- All of the job migration mechanisms described in the literature are either confined to very specialized platforms or of limited use in some applications. Specifically, in process and device simulation applications, no application independent checkpointing/job migration mechanism can hope to achieve satisfactory results.
- Many of the most popular global batch systems are specific to one particular operating system.
- Most job migration mechanisms are incapable of migrating applications involved in network communication.

Like the other systems, `DMW` fulfills most of the above criteria only partially:

- The installation of the `DMWMole` resource tracking daemon is intrusive, requiring superuser privileges to install it on the host computer. While this procedure is comparatively simple, it has repeatedly caused problems at customer sites, especially since certain NFS mounting options interfere with the execution of privileged programs.
- Since `DMW` has no job migration mechanism yet, it is occasionally obtrusive, prompting complaints by interactive users of hosts. While the literature is predominantly concerned with *CPU contention*, our experience has shown that for our applications, local schedulers are capable of handling CPU contention adequately, but that contention for physical *memory* (RAM) is a serious problem.
- While `DMW` runs on a wide range of POSIX based operating systems, it requires that sites have a common file namespace and mutually trusted user identities among the hosts. While these requirements are usually realistic for intranets, they rarely can be met in wide area networks.

8.1.2 Specific Design Objectives for `DMW`

Apart from the above limitations, `DMW` by now delivers good reliability and acceptable performance and fulfills most design objectives:

- Schedules get executed *correctly*, with all data and priority dependences being respected.
- The scheduler works on all target systems requested in 2.3, and it works with arbitrary heterogeneous clusters of such systems.

- The scheduler works with arbitrary applications.
- The scheduler maintains an accurate assessment of *load conditions* on the target systems, it correctly picks the least loaded hosts to schedule jobs on, and it will not schedule new jobs on overloaded systems.
- The scheduler maintains global *fairness* between clients executing schedules: All clients will tend to get the same number of concurrent jobs, and the same number of jobs over time.
- Schedules make constant *progress* and run to *completion* within a reasonable amount of time.

8.1.3 Fully Distributed vs. Centralized Schedulers

While the original implementation of DMW was fully distributed, the revised implementation introduced a central *arbitrator*. This change resulted in a significantly improved system:

- The arbitrator is able to guarantee *fairness* among clients.
- The centralized scheduler is not susceptible to *host contention*, which is a serious problem for the distributed scheduler.
- The centralized scheduler spends less bandwidth and measurement effort to collect resource availability data than the distributed scheduler.
- With the centralized scheduler, it is easier to gain an overview of all currently running simulation projects.

The objections commonly raised against centralized systems do not outweigh these advantages:

- Despite the centralized load information gathering effort, the arbitrator does not turn into a performance bottleneck, given the speed of modern workstation servers. In contrast, the increased bandwidth required for a fully distributed load information gathering may represent a serious scalability problem.
- While the arbitrator *process* as a single point of failure needs some additional programming effort to ensure a reliable restart after a failure, this effort presents no particularly difficult problems.
- The importance of the vulnerability of the *host* that the arbitrator process runs on can be minimized by placing the arbitrator on a file server host, whose failure would disrupt all operations in the workstation cluster anyway.

Given these considerations, a centralized organization appears to be the best design for a global scheduler.

8.2 Future Directions

The most obvious major improvement to scheduling in DMW would be obtained from integrating a checkpointing and job migration mechanism into it, which would resolve many situations where the current scheduler performs inadequately:

- If a host becomes overloaded due to intensified interactive use or other, non-DMW computations, scheduler jobs can be suspended and, as soon as possible, resumed on other hosts.
- If the combined working sets of one or multiple jobs on a host start exceeding the amount of physical memory available, some of them can be suspended and rescheduled on a different host or at a later time. With the updated knowledge about such jobs' memory needs, improved placement decisions can be made.
- For the above reasons, initial placement of hosts can follow a more aggressive policy if migration is available to correct overly aggressive placement.
- If fast hosts become idle, jobs running on slower hosts can be migrated to them. This possibility, had it been available, would have shortened the total running time of the experiment in Figure 7.14 by at least one hour.

However, as the discussion in section 5.7.2 established, integrating an application independent checkpointing and migration mechanism into DMW would not deliver these benefits, since:

- The large process size of simulation jobs would create unreasonable demands on disk space and I/O bandwidth for a general purpose checkpointing system.
- All general purpose migration mechanisms currently available for practical use support only homogeneous migration, placing severe restrictions on migration possibilities in heterogeneous workstation clusters.

Therefore, an application specific, cross platform checkpointing and migration mechanism, as outlined in section 5.8, appears to be the most promising approach to job migration in DMW. However, such a mechanism would have to be implemented in close collaboration with the developers of the simulators themselves.

The measurements in section 7.3.3, Figures 7.2 to 7.8, suggest that there is also some potential to improve the *prediction* of *resource consumption* of jobs. However, in opportunistic workstation clusters, this is of limited value as it is impossible to predict longer term *resource availability* in such an environment. Therefore, it is preferable to facilitate the potential of the scheduler for later *correction* of suboptimal placement.

In short, to achieve the best possible performance and unobtrusiveness, a global scheduler for heterogeneous opportunistic workstation clusters needs:

- An *information policy* that provides an accurate status of resource availability and shortages, including statistics not only about CPU load, but also about memory availability.
- A centralized *placement policy* to ensure fairness among scheduler clients and to get the most up to date load information with the least expense of network bandwidth and measurement effort.
- A *transfer policy* that provides both initial placement and the possibility for placement corrections through an efficient, heterogeneous job migration mechanism.

Part IV

Appendixes

Leer - Vide - Empty

Appendix A

The GENESISe Environment

> Every engineer in the hall, designing these nanotechnological toasters and hair dryers, wished he could have Hackworth's job in Bespoke, where concinnity was an end in itself, where no atom was wasted and every subsystem was designed specifically for the task at hand.
>
> Neal Stephenson, *The Diamond Age*

GENESISe is the graphical front-end to the ISE semiconductor technology CAD (TCAD) system offering

- Hierarchical project management facilities.
- Graphical process mask and simulation flow editors.
- Control of running simulations.
- Parametrization of all input files.

A.1 Creating a Simulation Project

After GENESISe is started, it presents the user with the *main window* shown in Figure A.1, listing icons for the tools available to the user.

To start a new project, the user first creates the project directory in the *database editor* and then opens the *tool flow editor* shown in Figure A.2 to specify the basic sequence of process simulation, grid generation, device simulation, and visualization steps.

Next, the user defines the process simulation by using the *process flow editor* (Figure A.3) to specify the process steps and the *layout editor* (Figure A.4) to specify the process masks.

As a final step, after defining the input command files of the other steps in the tool flow, the user can use the *parameter editor* shown in Figure A.5 to define variable parameters for the simulation.

The final result of these editing steps is a *tree of simulation steps* as shown in Figure A.6. The same process is simulated with two different values for the implantation dose and implantation energy. As a consequence, the simulation will require 4 independent process and device simulations to be carried out.

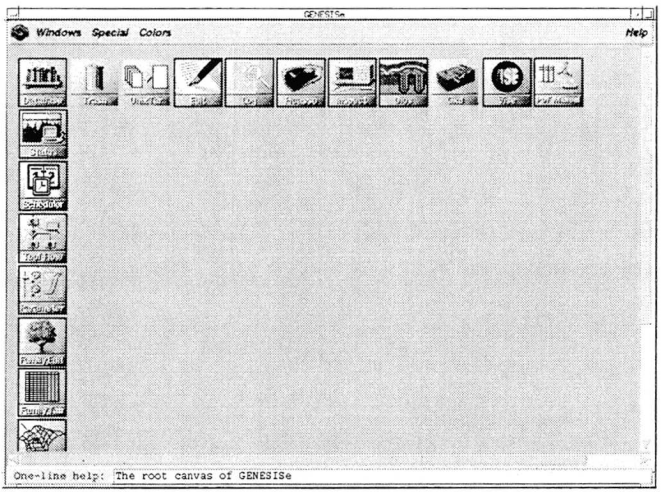


Figure A.1: GENESISe Main Window

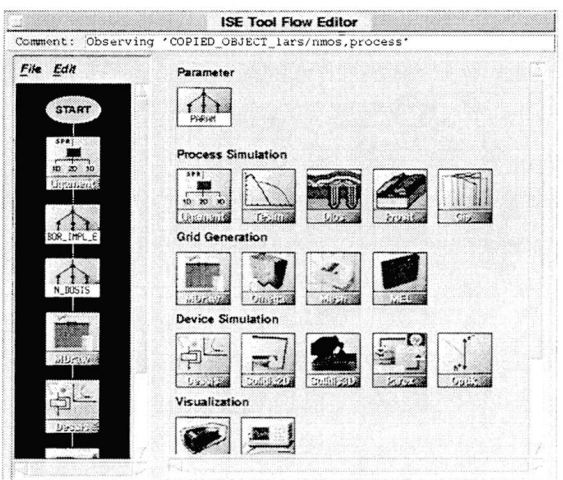


Figure A.2: GENESISe Tool Flow Editor

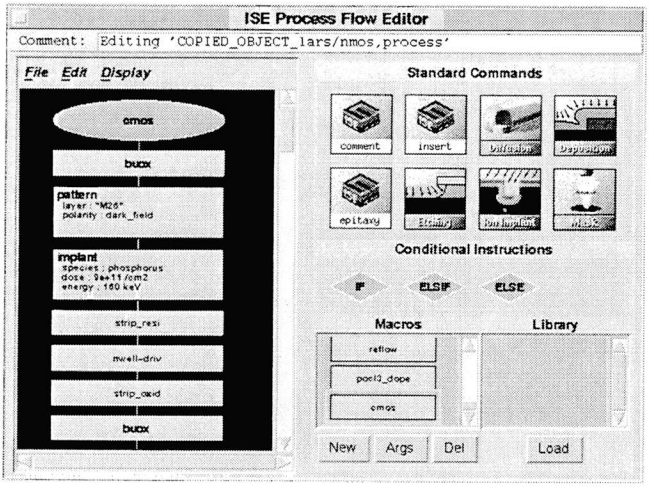


Figure A.3: GENESISe Process Flow Editor

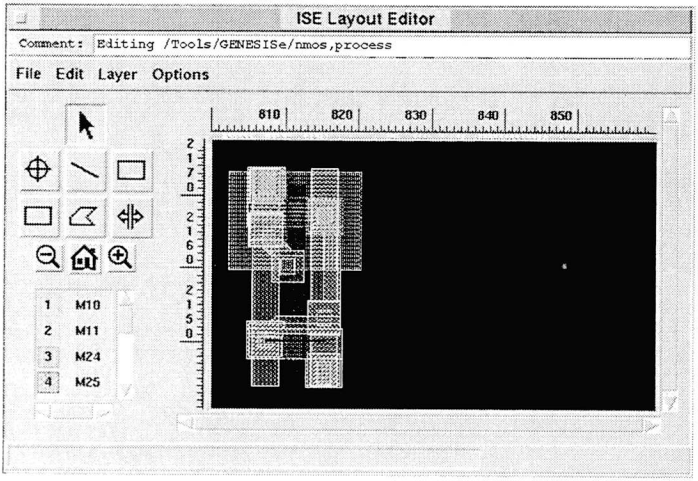


Figure A.4: GENESISe Layout Editor

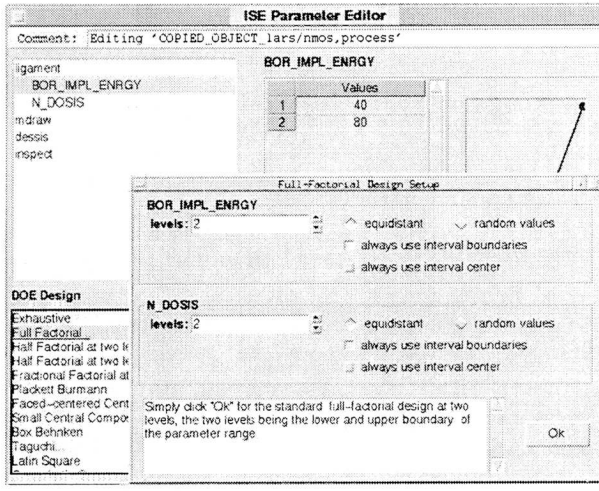


Figure A.5: GENESISe Parameter Editor

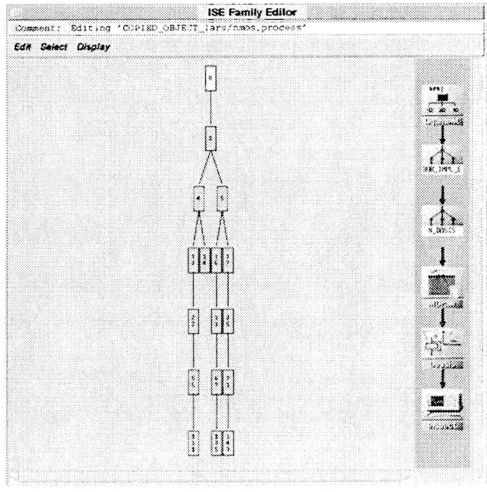


Figure A.6: GENESISe Simulation Tree

A.2 Executing the Project

After setting up the host and constraint databases in the *scheduler database window* shown in Figure A.7, the project can be executed: GENESISe creates a scheduler process and displays the status of the jobs in the simulation tree window (Figure A.6) and the status window (Figure A.8). The output log files generated by the simulations can then be accessed from the simulation tree window.

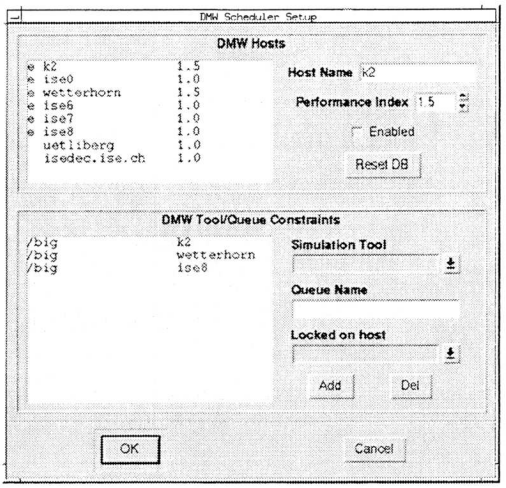


Figure A.7: GENESISe Scheduler Database

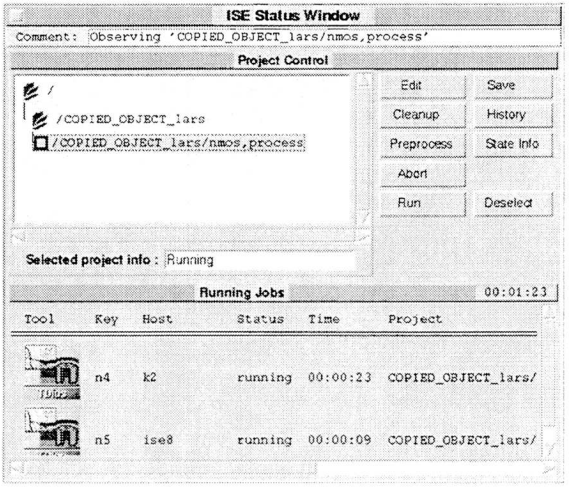


Figure A.8: GENESISe Status Window

Appendix B

A Glimpse at Queueing Theory

If I am given a formula, and I am ignorant of its meaning, it cannot teach me anything, but if I already know it what does the formula teach me?
Saint Augustine, *De Magistro*

Mathematically oriented papers about scheduling frequently assume some familiarity with statistics and queueing theory. In this appendix, mostly adapted from [Wol89] and [Pap84], I shall try to give a brief overview of this subject, concentrating on explaining frequently used terminology.

B.1 Events, Probabilities, Random Variables

A random experiment yields an *element* ω of the *sample space* Ω. An *event* A is a subset of Ω. A *probability measure* P is a real valued function satisfying the axioms:

$$P(A) \geq 0$$

$$P(\Omega) = 1$$

$$AB = \emptyset \Rightarrow P(A \cup B) = P(A) + P(B)$$

In the last case above, the events A and B are said to be *mutually exclusive*.

A *random variable* is a real-valued function $X(\omega)$ defined for every $\omega \in \Omega$. As a simple example of a random variable, an *indicator function* I_A of an event A is defined as

$$I_A(\omega) = \begin{cases} 1 & \text{if } \omega \in A \\ 0 & \text{if } \omega \notin A \end{cases}$$

The (cumulative) *distribution function* F of a random variable is defined as

$$F(x) = P(\{\omega : X(\omega) \leq x\})$$

which for simplicity is often written as

$$F(x) = P(X \leq x).$$

Instead of working with F, it is often convenient to work with the *tail distribution* of X, defined as $F^c(x) = 1 - F(x)$.

B.2 Continuous Distributions

If a random variable X has no values x with the property $P(X = x) > 0$, X is said to be *continuous* and there generally exists a *density function* $f(x) \geq 0$ such that

$$\begin{aligned} f(x) &= \frac{d}{dx}F(x) \\ F(t) &= \int_{-\infty}^{t} f(x)dx \end{aligned}$$

Three important density functions are those defining the *uniform distribution*

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise,} \end{cases}$$

the *normal distribution*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}},$$

and the *exponential distribution*

$$f(x) = \begin{cases} \mu e^{-\mu x} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0. \end{cases}$$

Finally, *hyperexponential distributions*, defined as

$$f_n(x) = \begin{cases} \sum_{i=0}^{n} w_i \mu_i e^{-\mu_i x} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0, \end{cases}$$

such that

$$\sum_{i=0}^{n} w_i = 1,$$

i.e., combinations of multiple exponential distributions, often provide a more accurate model of real queueing systems, but are harder to analyze.

B.3 Stochastic Processes

A *stochastic process* is an infinite collection of random variables $\{X(t) : t \in T\}$ defined over some *index set* T. Usually, t will denote *time* and T is defined as either $T = [0, \infty)$ for a *continuous-time* process or $T = 0, 1, \ldots$ for a *discrete-time* process. A particular point on the time axis is referred to as *epoch* t.

A *renewal process* counts the number of occurrences of some kind of event (e.g., the arrival of a bus at a bus stop) over time. It can be defined in terms of the epochs of the renewal events

$$\begin{aligned} Z_0 &= 0 \\ Z_n &= \sum_{j=1}^{n} X_j, \quad n = 1, 2, \ldots, \end{aligned}$$

where the X_j are independent, nonnegative random variables with distributions

$$\begin{aligned} P(X_1 \le t) &= A(t) \\ P(X_{j,j\ge 2} \le t) &= F(t). \end{aligned}$$

The renewal process itself is then a continuous-time stochastic process $\{M(t) : t \ge 0\}$ with

$$M(t) = \max\{n \ge 0 : Z_n \le t\}.$$

One frequently used case of a renewal process is the *Poisson process*, which is defined as $\{\Lambda(t)\}$ with

$$P\{\Lambda(t) = j\} = \frac{e^{-\lambda t}(\lambda t)^j}{j!}, \; j = 0, 1, \ldots$$

The interarrival times X_j are then distributed as

$$P(X_j > t) = e^{-\lambda t}.$$

B.4 Queueing Theory Concepts

Queueing theory models a stream of *customers* $C_1, C_2, \ldots$, numbered in their order of arrival. Each customer C_j arrives at some epoch t_j, waits in a *queue* if delayed by other customers, spends some time in *service* and on completion of service departs. Thus, we can define

$$\begin{aligned} S_j &= \textit{service time} \text{ of } C_j \\ D_j &= \textit{queue delay} \text{ of } C_j \\ W_j &= D_j + S_j = \textit{waiting time} \text{ in system of } C_j \\ t_j + W_j &= \textit{departure epoch} \text{ of } C_j \\ I_j(t) &= 1 \text{ if } t_j \le t < t_j + W_j, \text{ 0 otherwise} \\ N(t) &= \sum_{j=1}^{\infty} I_j(t) = \text{number of customers in system} \\ \Lambda(t) &= max\{j : t_j \le t\} = \text{number of arrivals by epoch } t \\ \Omega(t) &= \Lambda(t) - N(t) = \text{number of departures by epoch } t. \end{aligned}$$

In general, we are concerned with long-term behavior of a queueing model, and it is assumed that certain limits, defined as averages over time or customers, are constant over Ω or at least over a subset of Ω with probability 1. The most important such limits are

$$\begin{aligned} lim_{t\to\infty} \frac{\int_0^t N(u)du}{t} &= L, \quad \text{the } \textit{average number of customers} \\ lim_{t\to\infty} \frac{\Lambda(t)}{t} &= \lambda, \quad \text{the } \textit{arrival rate} \\ lim_{t\to\infty} \sum_{j=1}^{\infty} \frac{W_j}{n} &= w, \quad \text{the } \textit{average waiting time} \end{aligned}$$

These three limits are related through *Little's formula*

$$L = \lambda w.$$

B.5 Queueing Models

Distribution	A	B
Poisson (Exponential)	M	M
Constant (Deterministic)	D	D
General	GI	G
k-Erlang	E_k	E_k
Hyper-Exponential	H	H

Table B.1: Symbols used in $A/B/c$ queueing model notation

In the literature, queueing models are often characterized with the notation $A/B/c$, where c is the number of channels, A is the inter-arrival time distribution, and B is the service time distribution. Table B.1 shows the symbols used for specific distribution families in positions A and B.

A workstation cluster with K hosts and no global load distributing can then be described as a $GI/G/1$ or a $K \star GI/G/1$ system, while the ideal case of a perfect load distributing system with no overhead can be described as a $GI/G/K$ system. Usually, Poisson inter-arrival time distributions and exponential service time distributions are assumed, so system behavior is typically compared against $M/M/1, M/H/1, M/M/K$, or $M/H/K$ systems, but some of the literature (e.g. Wang and Morris [WM85]) discusses several other models.

Appendix C

Class Design Notation

> [T]he emancipation of the working classes must be conquered by the working classes themselves, [the] struggle for the emancipation of the working classes means not a struggle for class privileges and monopolies, but for equal rights and duties, and the abolition of all class rule [.]
>
> Karl Marx and Frederick Engels, *General Rules of the International Working Men's Association*

To illustrate compile-time and run-time relationships between C++ classes in Chapter 6, I use

Class Diagrams depicting classes, their structure, and the static relationships between them.

Interaction Diagrams showing the flow of requests between objects or between processes.

These diagrams were taken from the *Design Pattern* methodology pioneered by Gamma et al. [GHJV95]. Gamma et al. credit OMT (Object Modeling Technique) [RBP+91] for the class diagrams and Objectory [JCJO92] and the Booch method [Boo94] for the interaction diagrams.

C.1 Class Diagrams

Figure C.1 shows a class diagram with various relationships between the classes:

- Class inheritance is represented by a triangle connecting a subclass (LineShape in the figure) to its parent class (Shape).
- A part-of or *aggregation* relationship is indicated by an arrow with a diamond at the base. A filled circle at the tip of the arrow means that multiple objects are being aggregated (A Drawing contains several Shapes).
- An arrow without a diamond denotes *acquaintance* (e.g., a LineShape keeps a reference to a Color object, which other shapes may share).
- A dashed arrow indicates that one class instantiates objects of another (e.g. CreationTool creates LineShape objects).

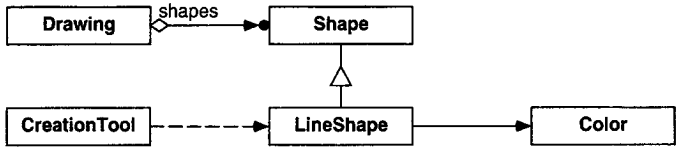


Figure C.1: Class Diagram Notation

C.2 Interaction Diagrams

Interaction diagrams show the order in which requests are executed. Figure C.2 is an interaction diagram of a shape getting added to a drawing.

Time flows from the top to the bottom of the diagram. A vertical solid line indicates the lifetime of a particular object, while a dotted line indicates that that object has not yet been instantiated.

A vertical rectangle shows that an object is currently handling a method call. The method can call methods of other objects; these are indicated with horizontal arrows pointing to the receiving objects, labelled with the method names. A request to create an object is shown with a dashed horizontal arrow.

In the example in Figure C.2, `aCreationTool` first creates `aLineShape`. Later, `aLineShape` is added to `aDrawing` which points the drawing to call its own `Refresh()` method. The `Refresh()` method in turn calls the `Draw()` method of `aLineShape`.

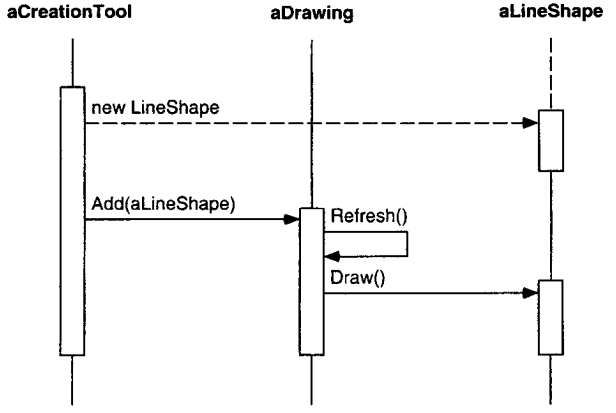


Figure C.2: Interaction Diagram Notation

Bibliography

> Les murs du cabinet de travail, le plancher, le plafond même portaient des liasses débordantes, des cartons démesurément gonflés, des boîtes où se pressait une multitude innombrable de fiches, et je contemplai avec une admiration mêlée de terreur les cataractes de l'érudition prêtes à se rompre.
>
> Anatole France, *L'île des pingouins*

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer 1986 USENIX Conf.*, pages 93–112, 1986.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[Ber86] M. J. Berglund. An introduction to the V-system. *IEEE Micro*, pages 35–52, August 1986.

[Blo92] J. Bloomer. *Power Programming with RPC*. O'Reilly & Associates, 1992.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[BS96] M. Buschauer and M. Steinmann. Entwurf eines Prozess-Migrations-Mechanismus. Technical report, Integrated Systems Laboratory, ETH Zurich, July 1996.

[CB94] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.

[CER95] CERT. NCSA HTTP daemon for UNIX vulnerability. Technical Report CA-95.04, CERT Coordination Center, February 1995.

[Cha93] S. J. Chapin. *Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems*. PhD thesis, Purdue University, December 1993.

[CIRM93] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: an object oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. on Software Eng.*, 14(2):141–154, February 1988.

[Com91] D. E. Comer. *Internetworking with TCP/IP*, volume I. Prentice Hall, 1991.

[Den68a] P.J. Denning. Thrashing: Its causes and prevention. In *Proc. AFIPS National Computer Conf.*, pages 915–922, 1968.

[Den68b] P.J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, 1968.

[DI89] M.V. Devarakonda and R.K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE Trans. on Software Eng.*, 15(12):1579–1586, December 1989.

[Dou89] F. Douglis. Experience with process migration in Sprite. In *Proc. Workshop on Experience with Distributed and Multiprocessor Systems*, October 1989.

[ELZ86] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Eng.*, 12(5):662–675, March 1986.

[Esk89] M.R. Eskicioğlu. Design issues of process migration facilities in distributed systems. *IEEE Technical Committee on Operating Systems Newsletter*, 4(2):3–13, Winter 1989.

[Feu98] T. Feudel. Personal Communication, February 1998.

[FV93] D. Farmer and W. Venema. Improving the security of your site by breaking into it. FTP Site, 1993. ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking-101.Z.

[FYN88] D. Ferguson, Y. Yemini, and C. Nikolau. Microeconomic algorithms for load balancing in distributed computer systems. In *Proc. IEEE 8th Int. Conf. on Distributed Computing Systems*, pages 500–508, 1988.

[FZ88] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Proc. Performance '87, the 12th Int. Symp. on Computer Performance Modeling, Measurement, and Evaluation*, pages 515–528, 1988.

[GC94] B. Goodheart and J. Cox. *The Magic Garden Explained.* Prentice Hall, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Gre94] T. P. Green. DQS 3.0 user guide. Technical report, Supercomputer Computations Research Institute, Florida State University, April 1994.

[GSS89] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A distributed batching system for parallel processing. *Software–Practice and Experience*, 1989.

[Hen95] R. L. Henderson. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing: IPPS '95 workshop*, pages 280–294, April 1995.

[Höf97] A. Höfler. *Development and Application of a Model Hierarchy for Silicon Process Simulation.* PhD thesis, Eidgenössische Technische Hochschule Zürich, 1997.

[HP91] N. Hutchinson and L. Peterson. The X-Kernel: an architecture for implementing network protocols. *IEEE Trans. on Software Eng.*, 17(1):64–76, January 1991.

[IEE90] IEEE, editor. *IEEE Standard portable operating system interface for computer environments.* Number 1003.1-1988 in ANSI-IEEE-STD. IEEE, New York, 1990.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering—A Use Case Driven Approach.* Addison-Wesley, 1992.

[JP94] ed. J. Postel. Internet official protocol standards. Technical Report RFC 1600, Internet Engineering Task Force, March 1994.

[KC91] P. Krueger and R. Chawla. The stealth distributed scheduler. In *Proc. IEEE 11th Int. Conf. on Distributed Computing Systems*, pages 336–343, 1991.

[KL87] P. Krueger and M. Livny. The diverse objectives of distributed scheduling policies. In *Proc. IEEE 7th Int. Conf. on Distributed Computing Systems*, pages 242–249, 1987.

[Knu92] D. E. Knuth. *Literate Programming.* Number 27 in CSLI lecture notes. Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

[KP95] O. Kipersztok and J. C. Patterson. Intelligent fuzzy control to augment scheduling capabilitties of network queuing systems. In *Job Scheduling Strategies for Parallel Processing: IPPS '95 workshop*, pages 239–258, April 1995.

[LBRT97] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. In *Proc. 21st Workshop on Vector and Parallel Computing*, pages 36–40. SPEEDUP, March 1997.

[LeF96] W. LeFebvre. top—display and update information about the top cpu processes. Technical report, Group sys Consulting, August 1996.

[Lew91] D. Lewine. *POSIX Programmer's Guide.* O'Reilly & Associates, 1991.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—a hunter of idle workstations. In *Proc. IEEE 8th Int. Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[Mak94] J. Maki. A free AIX performance monitor. In *Joint G.U.I.D.E / SHARE Europe Conference*, October 1994.

[MBKQ96] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[Mic88a] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Internet Engineering Task Force, March 1988.

[Mic88b] Sun Microsystems. RPC: Remote procedure call, protocol specification, version 2. Technical Report RFC 1057, Internet Engineering Task Force, June 1988.

[NCR97] M. Neeracher, C. Clémençon, and R. Rühl. A distributed scheduler for the GENESISe simulation system. In *Proc. 21st Workshop on Vector and Parallel Computing*, pages 46–49. SPEEDUP, March 1997.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Pap84] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, second edition, 1984.

[PL95] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In *Job Scheduling Strategies for Parallel Processing: IPPS '95 workshop*, pages 259–278, April 1995.

[PP97] J. S. Plank and M. Puening. Checkpointing Java. WWW Page, 1997. http://www.cs.utk.edu/~plank/javackp.html.

[Rau96] F. Rauch. Porting ckpt_lib to different UNIX operating systems. Technical report, ISE Integrated Systems Engineering AG, October 1996.

[Ray96] E. S. Raymond. *The New Hacker's Dictionary*. MIT Press, third edition, 1996.

[Ray98] E. S. Raymond. The cathedral and the bazaar. WWW Page, 1998. http://sagan.earthspace.net/~esr/writings/cathedral-bazaar/.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RC96] E. T. Roush and R. H. Campbell. Fast dynamic process migration. In *Proc. IEEE 16th Int. Conf. on Distributed Computing Systems*, pages 637–645, 1996.

[SHK95] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. Introduction to scheduling and load balancing. In *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society, 1995.

[SKS92] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, December 1992.

[SRE+89] E. H. Spafford, J. A. Rochlis, M. W. Eichin, D. Seeley, T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The internet worm (entire issue). *Communications of the ACM*, 32(6), June 1989.

[Ste90] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[Ste94] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

[Str98] N. Strecker. Personal Communication, January 1998.

[Sys96] Internet Security Systems. Comprehensive enterprise network security assessment. Technical report, White Paper, 1996.

[Tan87] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.

[TH91] M. M. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In *Proc. IEEE 11th Int. Conf. on Distributed Computing Systems*, pages 18–25, 1991.

[TL88] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proc. IEEE 8th Int. Conf. on Distributed Computing Systems*, pages 112–122, 1988.

[vM98] U. von Matt. Personal Communication, January 1998.

[WM85] Y.-T. Wang and R. T. Morris. Load sharing in distributed systems. *IEEE Trans. on Computers*, 34(3):204–217, March 1985.

[Wol89] R. W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, 1989.

[WS95] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1995.

[Zay87] E. R. Zayas. Attacking the process migration bottleneck. In *Proc. 11th Int. Symp. on Operating System Principles*, December 1987.

[Zay91] E. R. Zayas. AFS-3 programmer's reference: Architectural overview. Technical Report FS-00-D160, Transarc Corporation, 1991.

[ZF87] S. Zhou and D. Ferrari. An experimental study of load balancing performance. Technical Report UCB/CSD 87/336, Computer Science Division (EECS), University of California - Berkeley, January 1987.

[Zim98] R. Zimmermann. Personal Communication, January 1998.

Leer - Vide - Empty

Glossary

Rejected so far were: *an almanac* (too late in the year), *a breviary* (pointless), *a cadaster* (too bourgeois), *an encyclopedia* (would take too long), *a fescennine verse-dialogue* (only Lemprière knew what it was), *a glossary* (too many already), *an homily* (no), *incunabula* (too late), *juvenilia* (also too late), *a kunstlerroman* (too early), *a log* (Lemprière hated boats), *a manual* (boring), *a novel* (too vulgar), *an opera* (over-ambitious), *a pamphlet* (too humble), *a Qu'ran* (already was one), *a replevin* (too arcane), *a story* (too simple), *a treatise* (perhaps, but little enthusiasm), *an Upanishad* (too fanciful), *a variorum edition* (of what?), *a weltanschauung* (onanistic), *a xenophontean cosmology* (out of date) *and a year-book.*

Lawrence Norfolk, *Lemprière's Dictionary*

1D simulation — A simulation of process steps applied homogeneously to an infinite plane. Since the resulting material distribution is identical everywhere, only a single line intersecting the device needs to be simulated.

2D simulation — A simulation of process steps where a cross-section of a device is simulated.

3D simulation — A simulation of process steps where all dimensions of a device are simulated.

arbitrated scheduling — Nodes where jobs enter the system delegate placement decisions to a central arbitration process.

background job — A job assigned to a host by the distributed scheduler.

daemon — A program carrying out work in the background without user intervention.

dedicated cluster — A workstation cluster exclusively used for computationally intensive background tasks with no interactive work.

design pattern — A description of communicating objects and classes that are customized to solve a general design problem in a particular context [GHJV95].

device simulation — A numerical simulation of electrical current flow through a VLSI semiconductor device.

distributed scheduling Each node where jobs enter the system independently decides where the jobs are to be placed.

floating license A license limiting the total number of copies of a program running simultaneously on a group of hosts.

foreground job A job started on a host by an interactive user.

global scheduling Apportioning jobs among the hosts of a distributed system.

high throughput computing An approach that, as opposed to high *performance* computing, does not focus on maximizing peak performance, but on delivering large amounts of processing capacity over very long periods of time [LBRT97].

host A separate computer with its own set of IP addresses and its own local scheduler. A two-processor workstation, for instance, is treated as one host in this work.

individual scheduler A component of the scheduling system making placement decisions for one user.

information policy The set of host load information that is available to the scheduler.

internet A collection of packet switching networks interconnected by gateways along with protocols that allow them to function logically as a single, large, virtual network. When capitalized, Internet refers specifically to the connected global Internet and the TCP/IP protocols it uses [Com91].

intranet A TCP/IP based internet that has no connectivity or only highly restricted connectivity to the global Internet.

job An autonomous program executing in its own protection domain.

job migration Selecting a suitable host for a running job, suspending it on the current host, and resuming it on the new host.

job placement Selecting a suitable host for a job that hasn't started yet and starting the job there.

load balancing Distributing jobs, attempting to equalize the loads on all computers.

load sharing Distributing jobs among hosts in order to maximize the rate at which the distributed system performs work.

local scheduling Planning the assignment of a host to the jobs running on it.

multicasting Sending a message to a subset of hosts on a network.

nice value A scheduling parameter to lower the CPU allocation priority of a process (higher nice values for a process result in a smaller share of CPU time). On typical UNIX systems, nice values range from 0 to 19.

node locked license A license limiting the execution of a program to a particular group of hosts.

opportunistic cluster A workstation cluster using the spare computing resources of interactively used desktop workstations for computationally intensive background tasks.

page fault The user program attempts to reference a memory address that is part of its memory space, but not in physical memory at the moment. This causes the CPU to generate a hardware interrupt, and the operating system will load the memory *page* (a block of memory, typically a few kilobytes in size) containing the address from disk.

page replacement policy How an operating system selects pages to be removed from main memory.

placement policy Deciding which host in a distributed system a job will be transferred to.

platform A combination of hardware and operating system such that all hosts based on this platform can be expected to run software interchangeably.

port An abstraction used to distinguish among multiple destinations using the same protocol on a given host. TCP/IP protocols identify ports using small positive integers [Com91].

privileged software Programs that need to have access to files or services normally denied to the users running them. Privileged programs need to be installed by a system administrator, as their installation itself requires superuser privileges.

process simulation A numerical simulation of a VLSI semiconductor fabrication process.

receiver initiated scheduling Lightly loaded hosts search for heavily loaded hosts and solicit jobs from them.

residual dependency An on-going need for a host to maintain data structures or provide functionality for a process even after the process migrates away from the host [Dou89].

sender initiated scheduling Heavily loaded hosts search for lightly loaded hosts and transfer jobs to them.

socket An abstraction, introduced in Berkeley 4BSD UNIX and widely adopted in later UNIX systems, representing a network communication endpoint. Applications create a socket, connect it to a peer socket, and then send or receive data.

superuser privileges Ultimate access privileges on UNIX systems, granted to a special user ID, the superuser. The superuser, or software with superuser privileges, may access all services and any files, except in some cases on file systems mounted from a remote server by NFS.

transfer policy The conditions under which a job is transferred to a different host.

VLSI Very Large Scale Integration: Integrated Circuits with 10^5 or more components per chip.

working set The subset of a job's address space recently used and expected to be used in the near future.

workstation cluster A group of workstations, connected by a local area network.

XDR External Data Representation, a standardized specification for portable data transmission defined by Sun.

Index

But what I really like is the feeling of security I get from my new filing system; I have made myself more complicated than I really am.
Nick Hornby, *High Fidelity*

— E —

— F —

— G —

— H —

— I —

— J —

— K —

— L —

— M —

— R —

— S —

Leer - Vide - Empty

Curriculum Vitae

He was found by the Bureau of Statistics to be
One against whom there was no official complaint
W. H. Auden, *The Unknown Citizen*

I was born in Zürich, Switzerland, on February 1, 1967. In 1985, I graduated from high school at the *Kantonsschule Solothurn* with a Matura A/B and enrolled in computer science at *ETH Zürich* (Swiss Federal Institute of Technology). After receiving a MSc in computer science (*Dipl. Inf.-Ing. ETH*) in 1991, I joined the Integrated Systems Laboratory of ETH, where I worked as a research and teaching assistant. In addition to the work on job scheduling presented in this dissertation, I worked on automatic parallelization of Fortran code for distributed memory parallel processors and was involved in the design of two ASICs for a spread spectrum communication system.